**EMILIO EIJI YAMANE**

# MODELAGEM E IMPLEMENTAÇÃO DE UMA FERRAMENTA DE AUTORIA PARA CONSTRUÇÃO DE TUTORES INTELIGENTES

**FLORIANÓPOLIS**
**2006**

**UNIVERSIDADE FEDERAL DE SANTA CATARINA**
**CURSO DE PÓS-GRADUAÇÃO EM ENGENHARIA ELÉTRICA**

# MODELAGEM E IMPLEMENTAÇÃO DE UMA FERRAMENTA DE AUTORIA PARA CONSTRUÇÃO DE TUTORES INTELIGENTES

Dissertação submetida à
Universidade Federal de Santa Catarina
como parte dos requisitos para a
obtenção do grau de Mestre em Engenharia Elétrica.

**EMILIO EIJI YAMANE**

Florianópolis, Julho de 2006.

# MODELAGEM E IMPLEMENTAÇÃO DE UMA FERRAMENTA DE AUTORIA PARA CONSTRUÇÃO DE TUTORES INTELIGENTES

Emilio Eiji Yamane

'Esta Dissertação foi julgada adequada para a obtenção do título de Mestre em Engenharia Elétrica, Área de Concentração em *Controle, Automação e Informática Industrial*, e aprovada em sua forma final pelo Programa de Pós-Graduação em Engenharia Elétrica da Universidade Federal de Santa Catarina.'

_____

Prof. Guilherme Bittencourt
Orientador

_____

Prof. Nelson Sadowski
Coordenador do Programa de Pós-Graduação em Engenharia Elétrica

Banca Examinadora:

_____

Prof. Guilherme Bittencourt
Presidente

_____

Prof(a). Clara Amelia de Oliveira

_____

Prof(a). Jacqueline Gisèle Rolim

_____

Prof. Ricardo José Rabelo

*A todos aqueles que começaram no maravilhoso mundo da computação jogando num velho Atari (ou outro console qualquer)…*

# AGRADECIMENTOS

Ao meu pai Humberto e minha mãe Aparecida.

À minha irmã Cintia, que ficou me enchendo para ter o seu nome nestes agradecimentos.

Ao meu orientador prof. Guilherme, e as outras participantes do projeto que muito me ajudaram, Luciana e Eliane.

Ao meu orientador prof. Guilherme, e as outras participantes do projeto que muito me ajudaram, Luciana e Eliane.

Ao Takashi Hashiguchi-sensei, por mostrar as maravilhas do mundo do pão.

Às meninas do CLAMP, por criarem a mais bela flor de cerejeira que eu já vi.

A todo(a)s mangakás e pessoal de animação, por proporcionarem o meu passatempo e paixão.

E por fim, a todos os pesquisadores de redes, pois sem eles, como é que eu conheceria algumas das pessoas e suas obras, acima citadas...

Resumo da Dissertação apresentada à UFSC como parte dos requisitos necessários para obtenção do grau de Mestre em Engenharia Elétrica.

# MODELAGEM E IMPLEMENTAÇÃO DE UMA FERRAMENTA DE AUTORIA PARA CONSTRUÇÃO DE TUTORES INTELIGENTES

**Emilio Eiji Yamane**

Julho/2006

Orientador: Guilherme Bittencourt
Área de Concentração: Controle, Automação e Informática Industrial
Palavras-chave: Sistemas Tutores Inteligentes, Inteligência Artificial
Número de Páginas: xiii + 105

Este trabalho apresenta soluções de problemas na implementação de Sistemas Tutores Inteligentes (STI) no âmbito do projeto MathTutor, um arcabouço para construção de STIs.

São mostrados os fundamentos do MathTutor, envolvendo a tecnologia/paradigma de sistemas multiagentes, bem como a modelagem e implementação do módulo pedagógico utilizando Redes de Petri Objeto e Sistemas Especialistas, além do módulo de interface do aprendiz, via interface Web utilizando a tecnologia Servlets de páginas dinâmicas.

Abstract of Dissertation presented to UFSC as a partial fulfillment of the requirements for the degree of Master in Electrical Engineering.

# MODEL AND IMPLEMENTATION OF AN INTELLIGENT TUTORING SYSTEM AUTHORING TOOL

**Emilio Eiji Yamane**

July/2006

Advisor: Guilherme Bittencourt
Area of Concentration: Control, Automation and Industrial Computing
Key words: Intelligent Tutoring Systems, Artificial Intelligence
Number of Pages: xiii + 105

This work presents some problems and it's solutions in implementations of Intelligent Tutoring Systems (ITS), in MathTutor project, an ITS shell.

The concepts of MathTutor are shown, as well as part of it's implementation, composed, among some other things, by: the multiagent systems technology/paradigm; the pedagogical model which uses Object Petri Nets and Expert Systems; and the student interface module, which is a Web interface implemented by dynamic pages with Servlets.

# Sumário

**1 Introdução** **1**

1.1 Objetivos e Evolução deste Trabalho . . . . . 2

1.2 Organização do Trabalho . . . . . 3

**2 Fundamentação Teórica** **4**

2.1 Sistemas Multiagentes . . . . . 4

2.2 Representação de Conhecimento . . . . . 7

2.2.1 Sistemas Especialistas . . . . . 9

2.2.2 Ontologias . . . . . 11

2.3 Sistemas Tutores Inteligentes . . . . . 12

2.3.1 Componentes de um STI . . . . . 13

2.3.2 STIs como Sistemas Multiagentes . . . . . 14

2.3.3 Ferramentas de autoria de STIs . . . . . 15

2.4 Redes de Petri Objeto . . . . . 16

**3 Técnicas e Ferramentas** **18**

3.1 Java . . . . . 18

3.2 JavaCC . . . . . 19

3.3 JADE . . . . . 20

3.3.1 Plataforma . . . . . 20

3.3.2 Agentes JADE . . . . . 21

3.3.3 Mensagens ACL . . . . . 23

3.3.4 Outros Recursos do JADE . . . . . 25
3.4 JESS . . . . . 25
3.4.1 Regras e Fatos . . . . . 25
3.5 Protegé . . . . . 27
3.6 Servlets . . . . . 28

**4 MathTutor** **32**
4.1 O modelo conceitual MATHEMA . . . . . 32
4.2 Modelagem do MathTutor . . . . . 33
4.3 Um STI para Fundamentos da Estrutura da Informação . . . . . 34

**5 Implementação** **38**
5.1 Um agente tutor em JADE . . . . . 39
5.2 Um compilador para transformar uma Rede de Petri Objeto em um sistema de regras 40
5.3 Outras regras: adicionando semântica complementar na RPO . . . . . 41
5.4 Interface com o Estudante: um agente em JADE e integração com Servlets . . . . . 42

**6 Conclusões** **45**
6.1 Trabalhos Futuros . . . . . 45

**A Gramática da descrição da Rede de Petri Objeto** **46**

**B Transformando uma Rede de Petri Objeto em um Sistema de Regras** **50**
B.1 Primeiro Passo - Análise Sintática . . . . . 50
B.2 Segundo Passo - Adicionar Semântica na Gramática . . . . . 52
B.3 Terceiro Passo - Implementar as Estruturas de Dados . . . . . 55
B.3.1 OpnClass . . . . . 55
B.3.2 OpnPlace . . . . . 55
B.3.3 OpnVariable . . . . . 56
B.3.4 OpnTransition . . . . . 56
B.4 Quarto Passo - Coordenar todos os passos anteriores . . . . . 60

**C Extensões da RPO - funções complementares em JESS** **62**

C.1 Comunicação da RPO com o ambiente externo . . . . . 62

C.2 Implementação da semântica da RPO de primeiro nível . . . . . 66

C.3 Implementação da semântica da RPO de segundo nível . . . . . 67

**D Implementação de um agente em JADE para gerenciar as RPOs de controle do modelo pedagógico** **77**

**E Implementação da interface Web de Usuário do sistema MathTutor: Agente em JADE e Servlets** **87**

E.1 Agente JADE . . . . . 87

E.2 Servlets . . . . . 91

# Lista de Figuras


1.1 Arquitetura do modelo MATHEMA . . . . . 2

2.1 Componentes de um STI . . . . . 13

2.2 Um exemplo de Rede de Petri (a) Início (b) Depois da Transição1 ter disparado (c) Depois da Transição2 ter disparado . . . . . 17

3.1 Arquitetura de referência de Plataforma de Agentes da FIPA . . . . . 20

3.2 *Containers* e Plataformas JADE . . . . . 21

3.3 Execução interna de um agente JADE . . . . . 22

3.4 Ciclo de vida de um agente definido pela FIPA . . . . . 24

3.5 O caminho de uma página HTML . . . . . 29

3.6 Arquiteturas Web . . . . . 30

4.1 Arquitetura do modelo MATHEMA . . . . . 33

4.2 Grafo de Pré-Requisitos para um currículo do STI de Fundamentos da Estrutura da Informação . . . . . 35

4.3 O Grafo de Pré-Requisitos com os Problemas visualizados . . . . . 36

4.4 Rede de Petri Objeto de segundo nível do Modelo Pedagógico . . . . . 36

5.1 Elementos Implementados . . . . . 38

5.2 Fluxo da definição de um curso - Interface de Autoria . . . . . 39

5.3 Diagrama de classes do agente StudentPetriNetsAgent . . . . . 40

5.4 Arquitetura da interface Web . . . . . 43

5.5 Diagrama de Seqüência do padrão de interações entre Usuário/Servlets/Agente . . . 43

5.6 Diagrama de Seqüência ilustrando interação entre dois agentes . . . . . 44

B.1 Diagrama de Classes do compilador RPO/JESS . . . . . 54

B.2 Diagrama de Seqüência do processo de compilação . . . . . 61

C.1 Diagrama de Classes do pacote br.ufsc.MathTutor.JessComm . . . . . 73

C.2 Diagrama de Seqüência de comunicação síncrona entre JESS e algum objeto externo 74

C.3 Diagrama de Sequência da Comunicação assíncrona entre JESS e objeto externo - parte2 . . . . . 76

D.1 Diagrama de Classes do pacote `br.ufsc.MathTutor.PetriNetsAgent` . . . . . 82

D.2 Diagrama de Seqüência do agente StudentPetriNetsAgent . . . . . 84

D.3 Diagrama de Estados do agente StudentPetriNetsAgent . . . . . 86

E.1 Diagrama de Seqüência das interações Servlets/Agente . . . . . 95

E.2 Comunicação entre Agentes . . . . . 96

E.3 Diagrama de Estados do pacote ServletAgent . . . . . 98

E.4 Diagrama de Estados contendo o fluxo de execução dos Servlets . . . . . 100

# Lista de símbolos

| | |
|---|---|
| *ACL* | Agent Communication Language - Linguagem de Comunicação de Agentes, definida pela FIPA |
| *AMS* | Agent Managment System, sistema que no padrão FIPA controla o acesso e uso da plataforma de agentes |
| *API* | Application Programming Interface - Interface de Programação de Aplicações, conjunto de componentes de software pronto para ser usado. |
| *AT* | Agente Tutor |
| *BNF* | Backus Naur Form, uma notação formal para descrição da sintaxe de uma determinada linguagem |
| *CAI* | Computer Aided Instruction - Instrução Assistida por Computador |
| *CGI* | Common Gateway Interface, especificação para criação de pãginas HTML dinâmicas |
| *CLIPS* | C Language Integrated Production System - Sistema de Produção Integrada à Linguagem C, ferramenta para produção de Sistemas Especialistas |
| *DF* | Directory Facilitator, sistema que no padrão FIPA provê as funcionalidades de registro de serviços pelos agentes |
| *FIPA* | Foundation for Intelligent Physical Agents, organização ligada à IEEE que promove padrões para a tecnologia de agentes |
| *HTML* | HyperText Markup Language - Linguagem de Marcação de HiperTexto, linguagem em que são escritas as páginas da Web |
| *HTTP* | HyperText Transfer Protocol - Protocolo de Transferência de HiperTexto, protocolo utilizado para "navegar" pelas páginas Web |
| *IA* | Inteligência Artificial |
| *IA − ED* | Inteligência Artificial Aplicada à Educação |
| *IHM* | Interface Homem-Máquina |
| *JADE* | Java Agent Development Framework, um *middleware* de desenvolvimento de sistemas multiagentes em Java |
| *JavaCC* | Java Compiler-Compiler, gerador de analisadores léxico/sintáticos |
| *JESS* | Java Expert System Shell - *Shell* de Sistemas Especialistas em Java, composto de um motor de inferência e uma linguagem de *script* |

| | |
|---|---|
| *JSP* | JavaServer Pages, tecnologia Java para criação de páginas dinânicas na Web |
| *JVM* | Java Virtual Machine - Máquina Virtual Java |
| *LGPL* | Lesser General Public License, tipo de licença para distribuição de software livre |
| *RdP* | Rede de Petri |
| *RPO* | Rede de Petri Objeto |
| $RPO - MP$ | Rede de Petri Objeto do Modelo Pedagógico, reflete estrutura de relações de pré-requisitos entre as UPs |
| $RPO - Pb$ | Rede de Petri Objeto do Problema, controla a interação do aluno com as UIs |
| *SATA* | Sociedade de Agentes Tutores Artificiais |
| *SE* | Sistema Especialista |
| *SMA* | Sistema Multiagentes |
| *STI* | Sistema Tutor Inteligente |
| *UI* | Unidade de Interação |
| *UML* | Unified Modeling Language, linguagem de modelagem de sistemas, largamente utilizado para softwares orientados a objeto |
| *UP* | Unidade Pedagógica |

# Capítulo 1

# Introdução

Uma tutor pode ser definido como um instrutor particular que ensina alguma habilidade ou conteúdo específico para um estudante individual. Sistemas tutores procuram simular este processo de tutoria através de máquinas, sobretudo computadores. Sistemas Tutores Inteligentes vão um passo além, procurando incorporar ao processo de tutoria via computadores, componentes geralmente tidos como "humanos" ou "inteligentes", tais como capacidade de adaptação ao aluno, algum grau de empatia com o estudante, etc.

Sistemas Tutores Inteligentes (STI) são sistemas que fazem uso de diversas técnicas de Inteligência Artificial (IA) para atingir o objetivo de ensinar a um aluno humano. Nos últimos anos, muitos STIs, diferentes entre si tanto em filosofias de concepção quanto em técnicas de implementação, foram propostos e implementados. No entanto, a dificuldade de criá-los (de forma eficiente) ainda é um desafio a ser vencido. Para tentar diminuir esse problema, arcabouços (conjuntos de ferramentas, normalmente com interfaces gráficas) de STIs, as chamadas *ferramentas de autoria*, vêm sendo desenvolvidos.

O desenvolvimento do sistema MathTutor é um exemplo deste esforço [1] [2]. O MathTutor é um projeto com objetivo de criar um arcabouço para a construção de STIs baseado no paradigma de sistemas multiagentes. Neste contexto, utilizamos o modelo conceitual MATHEMA, proposto por Costa [3]. Neste modelo, há uma sociedade de agentes tutores, cada agente contendo um curso ou tutorial dentro de si, sendo que cada curso é baseado numa visão particular sobre um determinado assunto a ser ensinado. Além disso, há os módulos de interface de autoria, que serve ao desenvolvedor do curso, e a interface do estudante, que é o próprio sistema tutor do ponto de vista do estudante. A arqutetura deste modelo encontra-se na figura 1.1.

O projeto MathTutor conta com outros pesquisadores, os quais desenvolvem diferentes partes/ aspectos do sistema. De fato, como mostrado em [1] e [2], um protótipo foi anteriormente construído. Entretanto, para tornar o sistema mais robusto e incorporar uma ferramenta de autoria, optou-se pela re-construção do sistema, partindo-se do zero, reaproveitando apenas os conceitos teóricos.

Outros trabalhos do projeto MathTutor envolvem cooperação de grupos e sistemas adaptativos, sendo estes módulos desenvolvidos pelos outros pesquisadores do projeto. Os módulos apresenta-

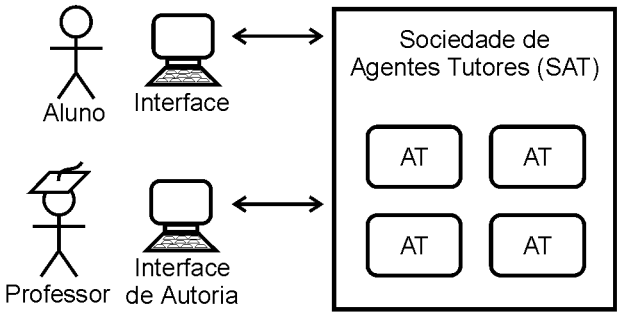



Figura 1.1: Arquitetura do modelo MATHEMA

dos neste trabalho servirão como base computacional para a implementação dos demais projetos de pesquisa.

## 1.1 Objetivos e Evolução deste Trabalho

Tendo já definidos o modelo e a arquitetura do MathTutor, o presente trabalho teve como objetivo a concepção, implementação e integração de diferentes módulos necessários para o sistema MathTutor. Para atingir este objetivo foram pesquisadas e comparadas diversas técnicas e ferramentas computacionais que poderiam ser utilizadas para suprir as funcionalidades exigidas.

Em especial, dois pontos do sistema foram aprofundados nesta pesquisa:

- fluxo de definição de um curso, pelo professor/tutor.

O responsável pelo desenvolvimento de um curso/tutorial deve definir a sua estrutura através de um grafo de pré-requisitos. Este grafo, de maneira transparente ao professor, é transformado numa Rede de Petri Objeto (RPO), que é o modelo previamente definido para a representação do curso. Esta RPO será transformada num conjunto de regras que possibilitem a sua execução num sistema de regras, sistema este que compõe o "coração" de um agente tutor. Este é o caminho da definição de um curso, desde a elaboração pelo professor até a sua implementação dentro de um agente tutor, e que faz parte do módulo de interface de autoria da arquitetura apresentada.

- concepção básica da estrutura computacional dos agentes e suas interações.

Neste trabalho é apresentado uma implementação de um agente tutor, sua estrutura/arquitetura básica, contendo um motor de inferência executando as regras referentes a RPO do modelo do curso, bem como outras regras. Além deste agente, é mostrado outro agente, não tutor, que serve de elo de ligação entre agentes tutores e os módulos de interface com o estudante. Este módulos, baseados numa arquitetura acessível pela Internet/Web, são compostos por Servlets.

A princípio, neste trabalho desenvolvemos um compilador para transformar uma Rede de Petri Objeto num conjunto de regras que pudesse ser executada num motor de inferência de Sistemas Especialistas, no nosso caso, o JESS. Este compilador toma como entrada uma instância de uma gramática particular, definida pelo nosso grupo de pesquisadores do MathTutor, e transforma num conjunto de regras em JESS.

Tendo este compilador definido, desenvolvemos dois modelos de RPOs básicas, que conteriam um curso a ser ensinado. A partir deste momento, o foco foi para o desenvolvimento de um agente tutor em si. Além do estudo das técnicas e tecnologias sobre agentes, foi necessário o desenvolvimento de uma integração entre o agente e o motor de inferência JESS, que contém a RPO com o curso, além de outras regras de controle.

Após esta etapa, ainda desenvolvemos o básico do módulo de interface com o estudante, através de Servlets. Para que os Servlets pudessem "conversar" com a SATA, outro tipo de agente foi necessário.

## 1.2 Organização do Trabalho

Este trabalho está organizado em seis capítulos (incluindo esta introdução), além de alguns apêndices.

No segundo capítulo encontra-se a fundamentação teórica, contendo uma pequena discussão sobre Agentes, Representação de Conhecimento e Sistemas Tutores Inteligentes em geral. No terceiro capítulo são apresentadas algumas técnicas e ferramentas pesquisadas e que foram utilizadas para a implementação.

No quarto capítulo é apresentado o sistema MathTutor, um arcabouço para construção de Sistemas Tutores Inteligentes, que utiliza o modelo conceitual MATHEMA. Além disso, é apresentada a concepção de um STI para disciplina Fundamentos da Estrutura da Informação, ministrada pelo Departamento de Automação e Sistemas da Universidade Federal de Santa Catarina.

No quinto capítulo é discutida a implementação de alguns módulos do sistema MathTutor, e por fim, no sexto capítulo encontram-se as conclusões e as perspectivas futuras.

Os apêndices contém descrições mais detalhadas da implementação, visando ajudar a qualquer programador que queira se aprofundar nos aspectos de implementação, utilizando, sempre que possível, uma linguagem de fácil acesso e com detalhes relevantes do código-fonte.

# Capítulo 2

# Fundamentação Teórica

Neste capítulo são abordadas sucintamente as teorias utilizadas na construção do sistema MathTutor.

De início, são apresentados os sistemas multiagentes, que servem de base para o modelo conceito MATHEMA [3], no qual a arquitetura MathTutor é baseada (seção 4.1). Em seguida examina-se o campo da representação de conhecimento, com ênfase em Sistemas Especialistas e Ontologias, ambas usadas na modelagem e construção dos agentes do sistema. Além disso, é apresentada a área de Sistemas Tutores Inteligentes e arcabouços para a construção destes. Por fim, examina-se brevemente as Redes de Petri Objeto, utilizadas na modelagem do módulo pedagógico do MathTutor (ver seção 4.2).

## 2.1 Sistemas Multiagentes

Segundo Russell e Norvig [4], um Agente é um sistema capaz de perceber através de sensores as informações do ambiente onde está inserido e reagir através de atuadores.

Entretanto, existe uma enorme diversidade de definições de agentes. Muitos pesquisadores preferem definir os agentes pelas propriedades que estes devem possuir. Rabelo [5] define três propriedades fundamentais:

- (um certo grau de) autonomia para raciocinar e tomar decisões por sua própria iniciativa;
- capacidade de interagir com outros agentes/sistemas/humanos;
- (um certo grau de) independência para resolver um problema, isto é, o agente tem conhecimento sobre como resolver (pelo menos parte) de um problema.

Os agentes podem ser classificados de diversas maneiras:

- Hardware ou Software;
- Estacionários ou Móveis;
- Persistentes ou Temporários;
- Reativos ou Cognitivos.

Tradicionalmente, classificam-se os agentes em reativos ou cognitivos. Os agentes reativos têm como características:

- Comportamento simples, apresentando uma resposta estereotipada a um conjunto de estímulos externos, resposta geralmente obtida através de um algoritmo de rápida execução. Não existe qualquer raciocínio baseado numa avaliação global do estado do mundo que justifique o conjunto de ações que devem ser desenvolvidas.
- O conhecimento do agente é assim reduzido a simples coleções de estímulos/respostas (baixa complexidade). Não possui uma representação interna de si próprio, dos outros agentes e do mundo onde se insere.
- O comportamento inteligente do sistema não se localiza em nenhum agente em particular, mas emerge da interação coletiva dos seus agentes, geralmente em grande número e fortemente acoplados.
- Sistemas com este tipo de agente são eficientes no seu tempo de resposta; entretanto, mostram limitações quanto aos problemas que podem resolver, sobretudo os que envolvem conhecimento dependente de raciocínio.

Os agentes cognitivos (ou ainda deliberativos ou simbólicos):

- Possuem objetivos próprios, que em conjunto com o conhecimento sobre o estado do mundo e de si próprios, ditam o seu comportamento. O agente é capaz de raciocinar sobre estes dois fatores, sendo possível selecionar as ações que irá realizar para maximizar a sua utilidade ou atingir o seu objetivo. O agente é capaz de controlar o seu próprio comportamento.
- Possuem uma representação simbólica de si próprios (das suas capacidades e objetivos), dos outros agentes e do mundo (representação parcial, nos dois últimos casos).
- O comportamento inteligente do sistema está localizado em cada um dos agentes individuais. Nessa inteligência podem estar incluídas uma racionalidade individual e uma racionalidade social.
- Geralmente cada agente possui uma grande quantidade de recursos, necessários para o raciocínio (alta complexidade). Isso faz com que o sistema não possua um grande número de agentes.

Num sistema multiagente, cada agente trata de uma parte do problema. Entretanto, nem sempre o agente com um determinado sub-problema tem capacidade para resolvê-lo, ou mesmo "vontade", dada a autonomia deste. Nestes casos, o agente pode iniciar uma interação com outros agentes.

A interação com outros agentes implica que:

- Cada agente passa a ter (algum) conhecimento das atividades dos outros agentes.
- São conhecidos os objetivos globais, o que pode levar a ações de cooperação ou competição.

Existem dois tipos de cooperação:

- Partilha de informações: acontece quando um agente conclui sobre um dado item de informação; dado o conhecimento sobre os outros agentes, este verifica se há agentes interessados nesse tipo de informação, e envia para eles. Estes, por sua vez, podem usar essa informação, descartá-la ou guardá-la para posterior uso.
- Partilha de tarefas: acontece quando um agente decompõe uma tarefa e detecta que não pode ou não deseja realizar alguma(s) de suas sub-tarefas. Assim, tendo conhecimento sobre os outros agentes, verifica se há agentes capazes e dispostos a ajudá-lo. Os agentes do sistema interagem pela partilha de carga computacional para a execução de sub-tarefas de uma tarefa global.

Para que a interação entre os agentes possa ser realizada, deve haver algum meio pelo qual estes se comuniquem. Existem dois paradigmas quanto a comunicação entre agentes:

- Troca de Mensagens: neste paradigma, os agentes comunicam-se diretamente enviando mensagens assíncronas. Para isso, é preciso que os agentes tenham conhecimento do nome/endereço uns dos outros, no caso de endereçamento direto, ou então de um endereço/nome específico para difusão (local ou global). Além disso, as mensagens devem seguir um formato estabelecido, com um protocolo/linguagem específicos e de comum entendimento entre os participantes da comunicação.
- Quadro-negro: neste paradigma, os agentes não se comunicam diretamente, mas podem ler ou escrever alguma informação numa estrutura de dados compartilhada, geralmente persistente, denominada quadro-negro (*blackboard*). A medida que o número de agentes aumenta, o processo de gravação e recuperação das mensagens no quadro-negro pelos agentes pode se tornar demorado demais para um sistema de tempo real. Se o paradigma de troca de mensagens pode ser visto como análogo aos programas de mensagens instantâneas, o quadro-negro pode ser considerado como sendo um fórum de mensagens.

Dada a autonomia dos agentes, podem, por vezes, surgir conflitos. Um exemplo de conflito é quando dois ou mais agentes discordam de determinada informação, quanto ao seu grau de veracidade ou valor. Nestes casos, algum mecanismo de controle é necessário. Este controle pode ser tanto

centralizado quanto distribuído. Pode-se, por exemplo, ter um agente com um grau de responsabilidade maior, que possa resolver determinado conflito, ou ainda, mais frequentemente, os agentes podem entrar em processo de negociação.

A aplicação da tecnologia de agentes na concepção de Sistemas de Informação é justificada, segundo Bolzan e Giraffa [6], quando o problema possui as seguintes características:

- domínio envolve distribuição intrínseca dos dados, capacidade de resolução de problemas e responsabilidades;
- necessidade de manter a autonomia de subpartes, sem a perda da estrutura organizacional;
- complexidade nas interações, incluindo negociação, compartilhamento de informação e coordenação;
- impossibilidade de descrição da solução do problema *a priori*, devido à possibilidade de pertubações em tempo real no ambiente e/ou processos de negócio, de natureza dinâmica.

## 2.2 Representação de Conhecimento

Segundo Davis et al., [7] a representação de conhecimento pode ser melhor compreendida se a olharmos por cinco prismas, ou por cinco papéis que ela representa. Estes papéis são:

- Uma representação de conhecimento é fundamentalmente um substituto, isto é, substitui alguma coisa por uma representação sua; isto permite que uma entidade possa determinar as conseqüências futuras pensando e não agindo, ou seja, ela não precisa executar uma ação para saber quais as suas implicações.
- Uma representação de conhecimento é um conjunto de compromissos ontológicos, isto é, uma resposta para a questão: “em que termos eu deveria pensar sobre o mundo?”; esses compromissos agem como lentes sobre o mundo, fazendo com que partes deste estejam bem focadas, enquanto outras partes ficam borradas, ou seja, imprecisas.
- Uma representação de conhecimento é uma teoria fragmentada sobre o raciocínio inteligente; este papel vem do fato da concepção inicial da representação do conhecimento ser tipicamente motivada por um questionamento de como as pessoas raciocinam, ou alguma crença sobre qual o significado do raciocínio inteligente. Esta teoria pode ser expressa em termos de três componentes: a concepção fundamental da representação do raciocínio inteligente (ou “qual o significado do raciocínio inteligente”); o conjunto de inferências que a representação possibilita (ou “o que podemos inferir a partir do que sabemos”); e o conjunto de inferências que recomenda (ou “o que deveríamos inferir a partir do que sabemos”).
- Uma representação de conhecimento é, pragmaticamente, um meio para computação eficiente. O raciocínio realizado por máquinas não é nada mais do que um processo computacional, ou seja, para poder ser usada, uma representação de conhecimento deve passar por processamento

computacional. Portanto, questões quanto à eficiência computacional são partes centrais na representação de conhecimento.

- Uma representação de conhecimento é um meio para a expressão humana, isto é, uma linguagem que nós (humanos) usamos para expressar fatos, conceitos sobre o mundo.

De certa forma, todo programa de computador contém o conhecimento sobre um determinado problema a ser resolvido. O conhecimento está nos algoritmos que o programa emprega e nos procedimentos de decisão que determinam quais destes algoritmos empregar em determinada circunstância [1].

Entretanto, para o uso em determinados sistemas, o conhecimento deve estar representado de maneira explícita, e não implicitamente codificado em algum algoritmo. Nestes casos, é necessário o uso de algum formalismo.

Em tese, uma representação geral como a lógica seria suficientemente expressiva para representar qualquer tipo de conhecimento. No entanto, problemas de eficiência, facilidade de uso e a necessidade de expressar conhecimento incerto e incompleto levaram ao desenvolvimento de outros tipos de formalismos de representação de conhecimento [8]. Dentre os formalismos, aqui destacamos: lógica, redes semânticas, quadros (*frames*) e regras de produção.

- Lógica: é a base para a maioria dos formalismos de representação de conhecimento. Trabalha com a manipulação de variáveis lógicas que representam proposições. As proposições podem ser verdadeiras ou falsas e podem ser combinadas através de conectivos. Mesmo os formalismos não lógicos têm, em geral, seu significado formal descrito através de uma especificação lógica de seu comportamento [1].

- Redes Semânticas: é uma notação gráfica composta por nós interconectados. As redes semânticas podem ser usadas para representação de conhecimento, ou como ferramenta de suporte para sistemas automatizados de inferência sobre conhecimento baseado em herança. Geralmente os nós representam substantivos, adjetivos, pronomes e nomes próprios, enquanto os arcos são reservados basicamente para representar verbos transitivos e proposições. As redes semânticas têm sido bem sucedidas porque a maioria dos formalismos das redes semânticas têm um modelo muito simples de execução, possibilitando aos programadores construírem grandes redes e ainda ter uma boa idéia sobre quais consultas serão mais eficientes, porque é muito simples visualizar os passos do processo de inferência [9].

- Quadros (*Frames*): em geral, um quadro é uma coleção de atributos, chamados de *slots*, e valores, que descrevem alguma entidade do mundo. Os quadros integram conhecimento declarativo sobre objetos e eventos e conhecimento procedimental sobre como recuperar informações ou calcular valores. Os atributos também apresentam propriedades, que dizem respeito ao tipo de valores e restrições de número que podem ser associadas a cada atributo; essas propriedades são chamadas facetas. Assim como nas redes semânticas, uma das características dos frames é a possibilidade de que sejam criados novos subtipos de objetos que herdem todas as propriedades

da classe original. Essa herança é bastante usada tanto para a representação do conhecimento como para a utilização de mecanismos de inferência [9].

- Regras de Produção: são compostas por duas partes, uma condição e uma conseqüência, por exemplo, “se este exercício está errado, então deve ser refeito”. As vantagens em se usar regras é que são modulares, ou seja, encapsulam conhecimento e podem ser facilmente expandidas, além do conhecimento ser expresso de maneira mais fácil de ser compreendido.

### 2.2.1 Sistemas Especialistas

Sistemas Especialistas (SEs) são um ramo da inteligência artificial aplicada, desenvolvido pela comunidade de IA nos meados de 1970. A idéia básica por trás dos Sistemas Especialistas é passar o conhecimento de um especialista humano para um computador, possibilitando aos usuários a possibilidade de consultar o computador para conselhos específicos quando necessário.

Sistemas Especialistas fornecem meios poderosos e flexíveis para a obtenção de soluções para uma variedade de problemas que geralmente não podem ser resolvidos com outros métodos mais tradicionais. Portanto, seu uso está proliferando em muitos setores da sociedade e comunidade tecnológica, onde as suas aplicações são críticas no processo de suporte à decisão e resolução de problemas [10].

Apesar de atualmente os Sistemas Especialistas serem concebidos usando-se diversas metodologias e formalismos, como é mostrado em [10], ainda pode-se encontrar o termo Sistema Especialista quase como um sinônimo de Sistema baseado em regras (ou Sistema de Produção), pois historicamente, os SEs tipicamente utiliza(ra)m-se de Regras de Produção como formalismo para representação de conhecimento.

Segundo Lustosa [9], os Sistemas Especialistas, em geral, podem ser divididos em três partes: uma base de regras, uma memória de trabalho (que contém uma base de fatos) e um motor de inferência. A base de regras e a memória de trabalho são chamados de base de conhecimento. As memórias de trabalho de SEs devem respeitar um método de representação de conhecimento.

O conhecimento precisa ser organizado de uma maneira adequada para que o motor de inferência consiga tratá-lo convenientemente. O conhecimento em um sistema especialista consiste de fatos e heurísticas. Os fatos constituem as informações que estarão sempre disponíveis para serem compartilhadas pelo especialista do domínio. As heurísticas são regras práticas que caracterizam o nível da tomada de decisão do especialista em um domínio. Uma base de conhecimento pode ser vista como um conjunto de regras, cada qual podendo ser validada independentemente da estrutura de controle [1].

O motor de inferência controla a atividade do sistema. Esta atividade ocorre em ciclos, cada ciclo consistindo em três fases [8]:

1. Correspondência de dados, onde as regras que satisfazem a descrição da situação atual são

selecionadas. Nesta fase, o motor de inferência utiliza-se de uma determinada *estratégia de busca*.

2. *Resolução de conflitos*, onde as regras que serão realmente executadas são escolhidas dentre as regras que foram selecionadas na primeira fase, e ordenadas.

3. Ação, a execução propriamente dita das regras.

O "raciocínio" do motor de inferência pode ser de dois tipos: encadeamento para frente (*forward chaining*) ou encadeamento para trás (*backward chaining*). No encadeamento para frente, o lado esquerdo das regras é comparado com os dados atuais, e as regras que satisfazem as condições são disparadas, possivelmente inserindo novos dados/fatos no sistema. Isso equivale à estratégia humana de ir examinando os dados até chegar a alguma conclusão.

No encadeamento para trás, existe um objetivo a ser alcançado, geralmente um ou mais fatos verdadeiros constando na base de dados. Neste caso, o motor de inferência verifica se estes já constam na base de dados, e se esta condição for verdadeira, o raciocínio chegou ao seu resultado final. Senão, o motor de inferência verifica quais regras podem levar ao objetivo alcançado, ou seja, verifica o lado direito das regras. Se no lado esquerdo da regra as condições forem verdadeiras (ou seja, há fatos na base de conhecimento para que a regra possa disparar), o raciocínio chega ao seu resultado final. Senão, o motor de inferência realiza o mesmo processo, de maneira recursiva, com os fatos do lado esquerdo da regra como sendo o seu objetivo. Isso equivale a estratégia humana de ter uma hipótese e então procurar por dados que possibilitem a sua conclusão.

A chave para a desempenho de um SE está no conhecimento armazenado em suas regras e em sua memória de trabalho. Este conhecimento deve ser obtido junto a um especialista humano do domínio e representado de acordo com regras formais definidas para a codificação de regras no SE em questão. Isto divide um SE em duas partes: a ferramenta de programação que define o formato do conhecimento da memória de trabalho e das regras, além dos aspectos operacionais de sua utilização, e o conhecimento do domínio propriamente dito [8].

Devido a esta separação, atualmente os Sistemas Especialistas são desenvolvidos em geral a partir de Arcabouços de Sistemas Especialistas (ou *shells* de SEs). Estes *shells* são ferramentas que suportam todas as funcionalidades de um SE, restando ao desenvolvedor apenas adicionar o conhecimento do especialista do domínio da aplicação. Este tipo de ferramenta foi um dos fatores responsáveis pelo crescimento de implementações de SEs.

Entretanto, mesmo tendo o seu uso apresentado crescimento, os Sistemas Especialistas segundo Lustosa [9] ainda restringem-se somente a domínios específicos, sendo inviável pensar na implementação de sistemas que respondam e reajam sobre temas gerais utilizando bases de conhecimento. Sem mencionar o fato de que a representação do conhecimento necessário exigiria uma base extremamente extensa.

Em [10], Liao faz um *survey* da área de Sistemas Especialistas e os classifica em diversas categorias, analisando trabalhos publicados de 1995 até 2004. Nurminen et al [11] analisa alguns casos em

que sistemas especialistas tiveram sucesso em aplicações industriais por longos períodos de tempo. Todavia, o Sistema Especialista mais citado é o MYCIN [12], um dos primeiros SEs desenvolvidos, em meados dos anos 1970, e seu objetivo é prover conselho a respeito de diagnóstico e terapia de doenças infecciosas. Com cerca de 450 regras e desenvolvido em Lisp [13], o MYCIN se saiu melhor do que médicos novatos e não especialistas, ficando pouco abaixo na média de acertos dos médicos especialistas. Mesmo assim, devido a questões legais, na prática ele não é usado [14].

### 2.2.2 Ontologias

Uma conceituação é uma visão abstrata e simplificada do mundo que se deseja representar para algum propósito. Uma ontologia é uma especificação explícita de uma conceituação [15]. O termo vem da filosofia, onde uma ontologia trata da natureza do ser, ou seja, da realidade, da existência dos entes e das questões metafísicas em geral [16].

Para sistemas de IA, o que "existe" é o que pode ser representado. Quando o conhecimento de um domínio é representado num formalismo declarativo, o conjunto de objetos que podem ser representados é chamado de universo de discurso. Este conjunto de objetos, e as relações destes objetos que podem ser descritas, são representados num vocabulário em um programa que pode representar o conhecimento. Portanto, no contexto da IA, pode-se descrever a ontologia de um programa definindo-se o conjunto de termos representativos. Nesta ontologia, definições associam os nomes das entidades com textos legíveis por seres humanos, que descrevem o que os nomes significam, e axiomas formais que restringem a interpretação e o uso destes termos [15].

Segundo Oliveira et al [17], com a utilização de ontologias é possível definir uma infra-estrutura para integrar sistemas inteligentes no nível do conhecimento, trazendo grandes vantagens como:

- Colaboração: possibilitam o compartilhamento do conhecimento entre os membros interdisciplinares de uma equipe;
- Interoperação: facilitam a integração da informação, especialmente em aplicações distribuídas;
- Informação: podem ser usadas como fonte de consulta e de referência do domínio;
- Modelagem: as ontologias são representadas por blocos estruturados que podem ser reutilizados na modelagem de sistemas no nível de conhecimento;
- Busca baseada em ontologia: recuperar recursos desejados em bases de informação estruturadas por meio de ontologias. Desta forma, a busca torna-se mais precisa e mais rápida, pois quando não é encontrada uma resposta exata à consulta, a estrutura semântica da ontologia possibilita, ao sistema, retornar respostas próximas à especificação da consulta [17].

Gruber [15] propõe um conjunto básico de critérios para o projeto de ontologias:

- Claridade: uma ontologia deve comunicar efetivamente o significado pretendido dos termos definidos. As definições devem ser objetivas; devem ser formais, ou seja, independentes de contexto social ou computacional; completas (quando possível); e documentadas em linguagem natural.

- Coerência: uma ontologia deve ser coerente, inferências feitas devem ser consistentes com as definições; ou seja, se uma sentença que pode ser inferida dos axiomas contradizer uma definição ou exemplo informal, então a ontologia é incoerente.

- Extensibilidade: uma ontologia deve ser capaz de definir novos termos para usos especiais baseados no vocabulário existente, sem que seja requerido uma revisão das definições existentes.

- Mínima influência de código: a influência de código resulta quando as escolhas de representação são feitas puramente para conveniência da notação ou implementação. Estas influências devem ser minimizadas.

- Mínimo comprometimento ontológico: uma ontologia deve fazer o mínimo de alegações possíveis acerca do mundo sendo modelado, permitindo aos parceiros especializar e instanciar a ontologia livremente.

Um *survey* analisando várias ferramentas de projeto de ontologia pode ser encontrado em [18].

## 2.3 Sistemas Tutores Inteligentes

Uma definição de tutor é “um instrutor particular que ensina alguma habilidade ou conteúdo educacional específico para um estudante individual; tal interação um-a-um permite ao tutor aumentar os conhecimentos ou habilidades do estudante mais rapidamente do que em uma classe de aula” [19].

Sistemas não-humanos que possuam como objetivo o ensino não são novidade: nos anos 20 do século passado, Sidney L. Pressey desenvolveu uma máquina com múltiplas questões e respostas. Esta máquina apresentava um retorno imediato ao seu usuário, que saberia se havia errado ou não [20].

Desde Pressey e seu aparato mecânico, os sistemas evoluíram, e hoje os sistemas tutores são complexos sistemas computacionais, que incorporam técnicas de Inteligência Artificial (IA). Historicamente, como aponta Self [21], a pesquisa em STI começou em conjunto com a própria pesquisa em IA, e vem constantemente evoluindo, tanto em técnicas como em filosofias, como o próprio Self mostra em [22].

De acordo com Rosatelli, “os Sistemas Tutores Inteligentes constituem uma tentativa de implementar, num sistema computacional, os métodos tradicionais de ensino e aprendizado exemplificados por uma interação um-a-um (entre tutor e aluno). O tutoramento um-a-um permite que o aprendizado seja altamente individualizado e, conseqüentemente, permite um melhor resultado” [23].

Sendo os STIs derivados dos programas CAI (*Computed Aided Instruction*, ou Instrução assistida por computador) [24], representam uma parte significativa da IA-ED (Inteligência Artificial na Educação) e freqüentemente são considerados como se constituíssem toda a área ao invés de apenas uma parte dela, como de fato [23].

Um sistema de IA-ED é um sistema computacional para o ensino que tem algum grau de tomada de decisão autônoma em relação às suas interações com os estudantes. Esse processo de decisão é realizado durante as interações do sistema com os usuários e, geralmente o sistema precisa acessar vários tipos de conhecimento e processos de raciocínio para habilitar tais decisões a serem tomadas. A IA-ED é parte da IA aplicada. As fontes para a área de IA-ED vêm de várias direções: primariamente da ciência da computação, psicologia, e educação, mas também de muitos outros campos, os quais são os tópicos dos sistemas de IA-ED [23].

Além dos Sistemas Tutores Inteligentes, a IA-ED tem também outros paradigmas, como a instrução assistida por computador (CAI), micromundos, ambientes de aprendizado inteligentes e aprendizagem colaborativa apoiada por computador. Dentre estes paradigmas, o que difere os STIs é exatamente o processo de tutoramente feito pelo sistema. Em outras palavras, o STI modela o entendimento do aluno sobre um tópico e à medida que ele interage com o sistema realizando tarefas, compara o conhecimento do aluno com um modelo do que se espera deste. Existindo diferenças entre o aluno e o que é esperado dele, o sistema se adapta para auxiliar o aluno a compreender o conteúdo a ser aprendido. Entretanto, esses paradigmas como CAI, aprendizagem colaborativa, etc, não são mutuamente excludentes, como, por exemplo, mostram os trabalhos relacionando a aprendizagem colaborativa e STIs [25].

### 2.3.1 Componentes de um STI

Geralmente, a estrutura de um STI tem seu modelo dividido em três módulos: Módulo do Estudante, Módulo do Domínio e Módulo Pedagógico ou do Tutor. Além destes, um componente de Interface também é geralmente mostrado, como ilustra a Figura 2.1, adaptada de [6] e [22].

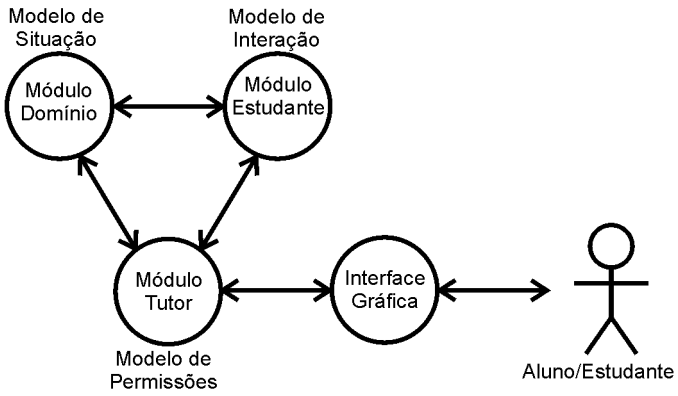



Figura 2.1: Componentes de um STI

O módulo do Estudante (também denominado do Aluno ou Aprendiz) armazena o modelo de um estudante individual. Este modelo representa o conhecimento do aluno (obviamente simplificado e

incompleto) num dado momento. A partir deste modelo, o sistema deve ser capaz de inferir o que realmente o estudante sabe, e qual estratégia de ação a ser tomada pelo STI.

O Modelo do Estudante, segundo Self em [22] deve focar no processo de interação, extenso no tempo, levando em consideração as ações do aluno, os contextos em que elas ocorreram, e as estruturas cognitivas do estudante nestes momentos. De fato, Self chama esse modelo de Modelo de Interação, do qual os Modelos do Estudante tradicionais (assim como foram primeiramente concebidos) são um subgrupo.

Na prática, a modelagem do Estudante (ou da Interação) ainda é um processo "artesanal" e que varia bastante entre os diversos STIs discutidos na literatura, como mostra Zhou em [26, 27], que faz uma análise de diferentes aspectos levados em conta na modelagem do Estudante num STI.

O Módulo do Domínio (também denominado do Conhecimento) contém a informação que está sendo ensinada. Segundo Bolzan et al [6], a modelagem do conhecimento a ser transmitido é de grande importância para o sucesso do sistema como um todo, devendo-se procurar uma representação do conhecimento que esteja preparada para o crescimento incremental do domínio.

Segundo Self [22], este módulo deve conter um Modelo de Situação. O Modelo de Situação é uma extensão do Modelo de Domínio comumente usado. Neste último, o construtor do STI tende a descrever o conhecimento em termos de fatos, princípios, etc., enquanto que no Modelo de Situação, o foco deve estar na natureza das situações, contextos e interações; ou seja, um modelo dos aspectos do conhecimento sobre o domínio [6]. Neste modelo o que existem são descrições dos recursos disponíveis numa situação de aprendizado, e não descrições do conhecimento que se pretende ensinar.

O Módulo Pedagógico (ou do Tutor ou Tutorial), segundo Bolzan et al [6] oferece uma metodologia para o processo de aprendizado. Possui o conhecimento sobre as estratégias e táticas para selecioná-las em função das características do aluno. As entradas deste módulo são fornecidas pelo Módulo do Aluno.

Neste módulo, segundo Self [22], temos um Modelo de Permissões, uma extensão do Modelo Pedagógico tradicional, no sentido de que o Modelo de Permissões visa não apenas modelar a seleção do conteúdo e estratégias, mas conduzir o aluno de acordo com objetivos e desafios educacionais, em termos de "itens de conhecimento", que podem ser aprendidos através de eventos particulares.

O Módulo de Interface é o módulo que controla a interação entre o tutor e o aluno. Neste módulo, encontramos uma variedade de implementações e estilos, variando desde simples janelas com elementos gráficos, até reconhecimento de linguagem natural (como por exemplo, em [28]), e desde sistemas com interface local até sistemas de ensino a distância, geralmente com interface tipo Web.

### 2.3.2 STIs como Sistemas Multiagentes

Segundo Rabelo [5], vive-se atualmente um momento de rápida expansão na utilização de Sistemas Multiagentes (SMA) na construção de modernos e complexos sistemas distribuídos, nas mais variadas áreas.

A área de STIs não é exceção, e nos últimos anos, cada vez mais arquiteturas com agentes vêm sendo pesquisadas, como pode-se ver no estudo de Bolzan e Giraffa em [6], que faz um estudo comparativo entre STIs desenvolvidos como Sistemas Multiagentes Web, além de outros exemplos como [29, 30, 31, 32].

Segundo Gurer [33], o uso de agentes na concepção de sistemas educacionais traz algumas vantagens, tais como: reagir às ações do usuário, credibilidade, modelagem de sistemas colaborativos multi-usuário e modularidade, pelo fato de que cada agente é um módulo único e independente do outro, ficando mais fácil adicionar outros agentes a estes sistemas [6].

Em [6] são mostradas as seguintes vantagens da abordagem multiagente:

- conhecimento pode ser distribuído entre vários "tutores", cada um com suas crenças, desejos, objetivos e planos de ação. Esta distribuição cria maiores oportunidades de variar técnicas pedagógicas;
- aprendiz interage com um tutor de forma mais flexível;
- aprendiz pode passar conhecimentos ao tutor que serão repassados a outros aprendizes.

### 2.3.3 Ferramentas de autoria de STIs

Segundo Murray [34], mesmo com os STIs cada vez mais se tornando comuns e com crescente eficiência no ensino comprovada, STIs ainda são difíceis e caros de serem construídos.

Para sistemas tradicionais de instrução por computador (CAI - computer aided instruction) e treinamento baseado em multimídia, existem vários *sistemas de autoria* (também chamados de sistemas de autoração). Entretanto, esses sistemas carecem de recursos para construção de STIs. De fato, estes sistemas focam nas ferramentas de construção de interface visuais atraentes, com telas interativas e variados recursos para o projeto destas, enquanto as ferramentas para auxliar na construção da representação do domínio e do modelo pedagógico são extremamente simples, quando existentes.

Ferramentas de autoria para STIs, portanto, são um passo importante para a disseminação de novos STIs, facilitando a produção destes com custos menos elevados. Além dos custos, do ponto de vista do professor ou autor do curso, a facilidade de desenvolvimento de um STI talvez seja um dos pontos mais cruciais no uso de ferramentas de autoria, afinal STIs são sistemas complexos e não triviais.

Ferramentas de autoria de STIs, ao contrário dos sistemas de autoração tradicionais, devem, além da facilidade de desenvolvimento do curso, lidar com a complexidade inerente aos STIs, complexidades estas que percorrem desde os campos de IA até campos da psicologia do ensino. Além disso, ao contrário das ferramentas de autoria tradicionais, onde a apresentação/interação com o usuário pouco sofre alterações, nas ferramentas de autoria para STIs a apresentação deve ser extremamente adaptável, refletindo as mudanças nos modelos do aluno e decisões baseadas no modelo pedagógico.

Uma análise sobre as ferramentas de autoria para STIs, bem como exemplos e classificações destas, podem ser encontrados em [34].

## 2.4 Redes de Petri Objeto

A Rede de Petri (RdP) é uma ferramenta gráfica e matemática que se adapta bem a um grande número de aplicações em que as noções de eventos e de evoluções simultâneas são importantes. Esta teoria nasceu da tese intitulada *Comunicação com autômatos*, defendida por Carl Adam Petri em 1962 na Universidade de Darmstadt, Alemanha. Entre as aplicações das Redes de Petri pode-se citar: avaliação de desempenho, análise e verificação formal em sistemas discretos, protocolos de comunicação, interface homem-máquina e multimídia [35].

Graficamente, uma RdP pode ser representada por um grafo com dois tipos de nós: lugares e transições. Os nós lugares são representados por círculos, e as transições por traços ou retângulos. Os arcos podem possuir pesos, ou seja, valores naturais positivos, e são sempre direcionados, ligando um lugar a uma transição ou uma transição a um lugar. Os nós lugares podem conter elementos chamados de fichas. Ao conjunto de fichas associadas a cada lugar, num dado momento, dá-se o nome de marcação.

Uma transição é dita sensibilizada, quando cada lugar que é conectado à transição por um arco lugar-transição (lugares de entrada da transição) tem um número de fichas maior ou igual ao peso do arco. Transições que estão sensibilizadas podem disparar, causando a retirada das fichas dos lugares de entrada correspondentes ao peso do arco, e ao mesmo tempo inserindo fichas nos lugares de saída (os lugares ligados a transição via arcos transição-lugar), correspondendo também aos pesos dos arcos. Um exemplo é mostrado na figura 2.2.

A definição formal de Redes de Petri, bem como maiores explicações e exemplos pode ser visto em [35].

Redes de Petri Objeto (RPO) são uma extensão das Redes de Petri, em que são usadas técnicas de modelagem orientadas a objeto, o que visa facilitar a modelagem de sistemas complexos. Lakos em [36] traz uma definição formal para as RPO, e em [37] faz uma apanhado informal das RPO, mostrando a evolução dos formalismos relativos as Redes de Petri, desde as Redes de Petri Coloridas até as RPO.

Nas RPOs as fichas podem representar objetos, ou seja, carregam consigo tanto estruturas de dados quanto métodos que podem ser chamados para atuar sobre algum de seus dados.

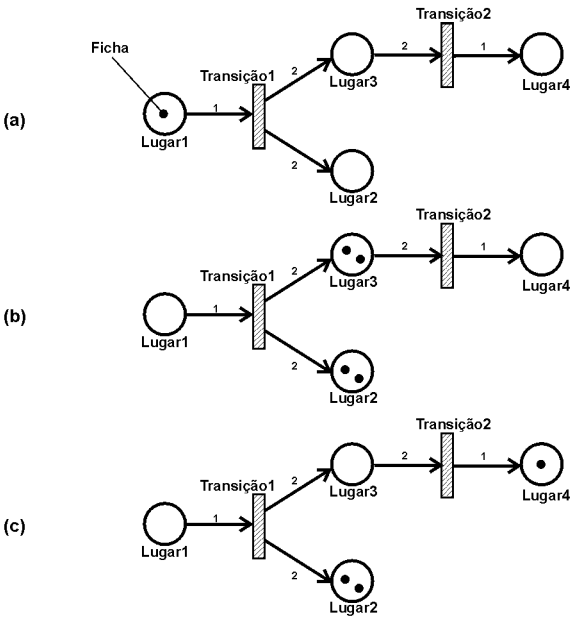



Figura 2.2: Um exemplo de Rede de Petri (a) Início (b) Depois da Transição1 ter disparado (c) Depois da Transição2 ter disparado

# Capítulo 3

# Técnicas e Ferramentas

Neste capítulo são apresentadas as ferramentas utilizadas na construção do MathTutor. A começar pela linguagem e plataforma Java, que é utilizada em todas as outras ferramentas.

Também é apresentado a ferramenta JavaCC, que gera analisadores sintáticos e léxicos em Java. Estes analisadores são usados na implementação de um compilador de Redes de Petri Objeto (seção 2.4) em um sistema de regras (descrito em 5.2).

Em seguida é apresentada a plataforma JADE, uma plataforma de suporte a sistemas multiagentes desenvolvida em Java, e que conterá os agentes tutores do MathTutor (ver seção 4.1). O *shell* de Sistemas Especialistas JESS é apresentado, bem como a ferramenta para ontologias Protegé. Ambas ferramentas utilizadas na modelagem e construção interna dos agentes tutores (ver 4.2).

Além disso, a tecnologia de páginas HTML dinâmicas Servlets, também é mostrada. Esta tecnologia é a base para a construção do módulo de interface Web do MathTutor (ver em 5.4).

## 3.1 Java

A tecnologia Java é composta por uma linguagem de programação e por uma plataforma de software. A linguagem de programação Java é orientada a objeto, simples e robusta ao mesmo tempo, e seu projeto inclui recursos como segurança, portabilidade, desempenho e suporte a redes e sistemas distribuídos. Maiores discussões sobre a linguagem Java e seus atributos, bem como um relato histórico de seu desenvolvimento, podem ser encontrados no trabalho de Gosling [38], um dos criadores da tecnologia Java.

Ao contrário de linguagens como C/C++, o código em linguagem Java não é compilado diretamente para código de máquina específico de um Sistema Operacional/Plataforma de Hardware, mas para um código intermediário conhecido como *bytecode*. Este *bytecode* é executado em uma máquina virtual (a JVM - Java Virtual Machine), ou seja, um software que “traduz” o *bytecode* para código de máquina real. Isso torna o código feito em Java portável para qualquer sistema em que haja uma

máquina virtual implementada, ou seja, o mesmo bytecode criado num sistema Windows executa da mesma maneira num sistema Linux, Unix, Solaris, ou outro qualquer, desde que haja a presença da máquina virtual. Além disso, as JVMs mais modernas também realizam análise de performance do código, executando, por exemplo, compilação do *bytecode* para código de máquina nativo de trechos críticos do código.

Além da máquina virtual, a plataforma Java é composta de uma API (Application Programming Interface), um conjunto de componentes de software, em Java, pronto para ser usado. A API do Java contém, por exemplo, classes direcionadas à construção de interfaces gráficas para o usuário, recursos de segurança em rede e sistemas distribuídos, além de vários outros recursos. Para detalhes da API, pode-se consultar a documentação existente no site do Java [39].

Por seus méritos, a tecnologia Java vem conquistando muitos pesquisadores e desenvolvedores, o que se reflete no número de projetos que utilizam o Java, muitos dos quais gratuitos ou *open-source*. Isto pode ser verificado com uma consulta ao site Java.net [40], que agrupa várias comunidades de entusiastas/desenvolvedores do Java, ou ao SourceForge [41], site que agrupa vários projetos open-source.

Devido às suas qualidades já citadas, bem como ao grande número de projetos e fortes laços com a filosofia *open-source*, Java foi escolhida como linguagem e plataforma de desenvolvimento do projeto MathTutor.

## 3.2 JavaCC

JavaCC (Java Compiler-Compiler) [42] é uma ferramenta gratuita desenvolvida em Java que gera analisadores sintáticos e léxicos em Java. Similar aos já bem conhecidos Lex e YACC para ambientes Unix, a vantagem é que o código gerado pelo JavaCC é muito mais simples de ser lido por um ser humano.

O JavaCC toma como entrada um arquivo de especificação de gramática. Este arquivo tem uma estrutura muito similar a notação BNF (Backus Naur Form) [43]. Além da descrição da gramática, o arquivo pode conter também código Java embutido entre as construções da gramática, o que provê grande flexibilidade para os analisadores gerados.

Um dos muitos exemplos do uso do JavaCC encontra-se em [44], um projeto de implementação da linguagem Python em Java.

Entretanto, o JavaCC não tem disponível muita documentação, excetuando-se os exemplos que acompanham o pacote de instalação e (num nível mais avançado) os repositórios de gramáticas disponíveis na Internet. Kodaganallur em [45] oferece um tutorial sobre o uso do JavaCC, com casos de uso do mesmo.

## 3.3 JADE

JADE (Java Agent Development Framework) [46] é um *middleware* para o desenvolvimento de aplicações distribuídas multiagentes, baseado na arquitetura de comunicação *peer-to-peer* [47].

JADE é inteiramente desenvolvido em Java, e segue os padrões da FIPA [48]. A FIPA (*Foundation for Intelligent Physical Agents*) é uma organização ligada a IEEE, que promove padrões para o desenvolvimento de agentes inteligentes e sistemas multiagentes. Isso possibilita que os agentes implementados em JADE se comuniquem com outros agentes que sigam o mesmo padrão. Além disso, JADE segue a filosofia *open-source*, sendo distribuído sob a licença LGPL [49].

Para mais detalhes sobre o JADE (sua arquitetura, funcionalidades e modelos conceituais), pode-se consultar [47]; para uma análise de desempenho da plataforma JADE, consulte [50]; além da página na Web [46], que contém links e trabalhos relacionados, bem como material atualizado da plataforma.

### 3.3.1 Plataforma

A plataforma JADE segue o padrão da FIPA, e é mostrado na figura 3.1 (adaptado de [51]).

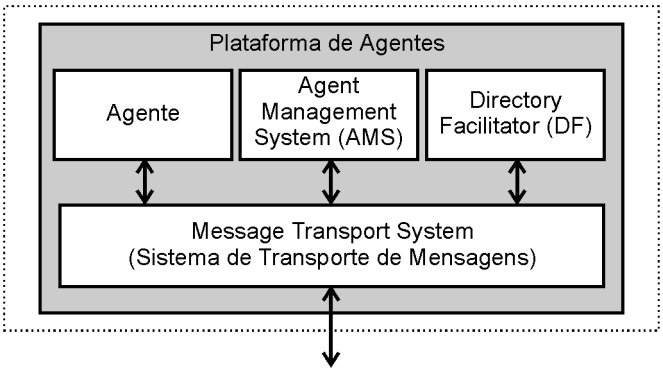



Figura 3.1: Arquitetura de referência de Plataforma de Agentes da FIPA

Nesta figura, pode-se observar o *Agent Management System* (AMS). O AMS é o agente que controla o acesso e uso da plataforma de agentes, e existe um único AMS em uma plataforma. O AMS provê serviços de gerenciamento de ciclo de vida dos agentes, serviços de busca, assim como serviço de nomes (mantendo um diretório de identificadores de agentes - AID), usados pelos agentes que devem se registrar com o AMS para obterem um AID válido.

O *Directory Facilitator* (DF) é o agente que registra os serviços oferecidos pelos agentes na plataforma, e que provê buscas por estes serviços. Ou seja, os agentes, ao entrarem na plataforma, podem registrar os serviços que oferecem e, quando necessário, podem consultar se há algum agente que ofereça determinado serviço.

O Sistema de Transporte de Mensagem (Message Transport System), também chamado de Canal

de Comunicação de Agente (Agent Communication Channel - ACC) é o componente de software que controla a troca de mensagens dentro da plataforma [51].

Tanto o AMS quanto o DF são agentes, e para interação com estes, necessita-se utilizar, nas comunicações, a linguagem FIPA-SL0, a ontologia de gerenciamento de agentes FIPA, e o protocolo de interação FIPA-Request. Entretanto, o JADE fornece, dentro de sua API, métodos que encapsulam muito desta complexidade.

Cada plataforma JADE pode ter vários *Containers*. *Containers* são os "locais" onde os agentes efetivamente residem. Cada *Container* reside em um host ou máquina virtual Java (JVM). O conjunto dos Containers forma uma plataforma. Cada plataforma tem um único *Container* especial, denominado de principal (Main Container), em que geralmente reside o AMS, além de ser responsável pelo registro dos outros Containers da plataforma. Caso outro *Container* seja iniciado como principal, este fará parte de outra plataforma lógica. Esta arquitetura é mostrada na figura 3.2 (adaptado de [52]).

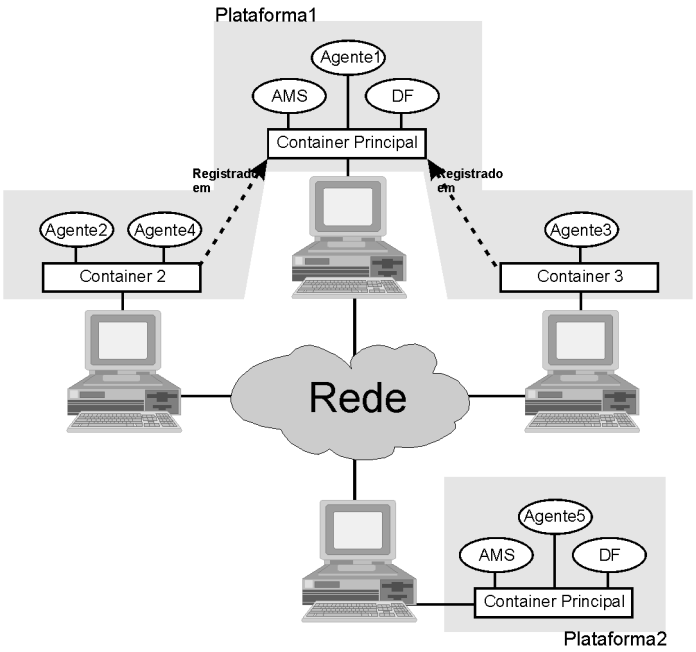



Figura 3.2: *Containers* e Plataformas JADE

### 3.3.2 Agentes JADE

Os agentes em JADE são, do ponto de vista da programação, objetos cujas classes extendem a classe padrão dos agentes. Várias funções já estão implementadas nesta classe padrão, o que poupa ao programador muitas tarefas comuns, sobretudo as interações básicas com a plataforma, como registro do agente junto ao AMS. Além disso, outras funções para recebimento/envio de mensagens, e suporte à programação das ações dos agentes também estão implementadas na classe padrão de agentes.

Os agentes em JADE utilizam o modelo de multitarefa. Em JADE, cada tarefa ou ação dos agentes deve ser implementada como um *Comportamento*, ou *Behaviour*. Para isso, é necessário que

se extenda uma das diversas classes de *Behaviour* implementadas, disponíveis na API. Feito isto, basta que se associe uma instância desta classe de *Behaviour* a um agente, para que o agente possa executar a ação programada.

Os diversos *Behaviours* são escalonados internamente, tendo cada agente, apenas uma thread de execução. Entretanto, diferentemente das *threads* Java, o escalonamento não é preemptivo, ou seja, uma vez o *Behaviour* sendo executado, ele só parará quando o seu método de ação for inteiramente executado. Isso traz como desvantagem o fato de que o programador deve estar atento, para que os *Behaviours* ajam de maneira fluida, mas traz como vantagem melhor performance, possibilitando uma única *thread* por agente, eliminando questões de sincronização ao acesso de recursos e eliminando a mudança de contexto entre diferentes *threads* [52]. A figura 3.3 (adaptado de [52]) ilustra o processo interno de um agente em JADE.

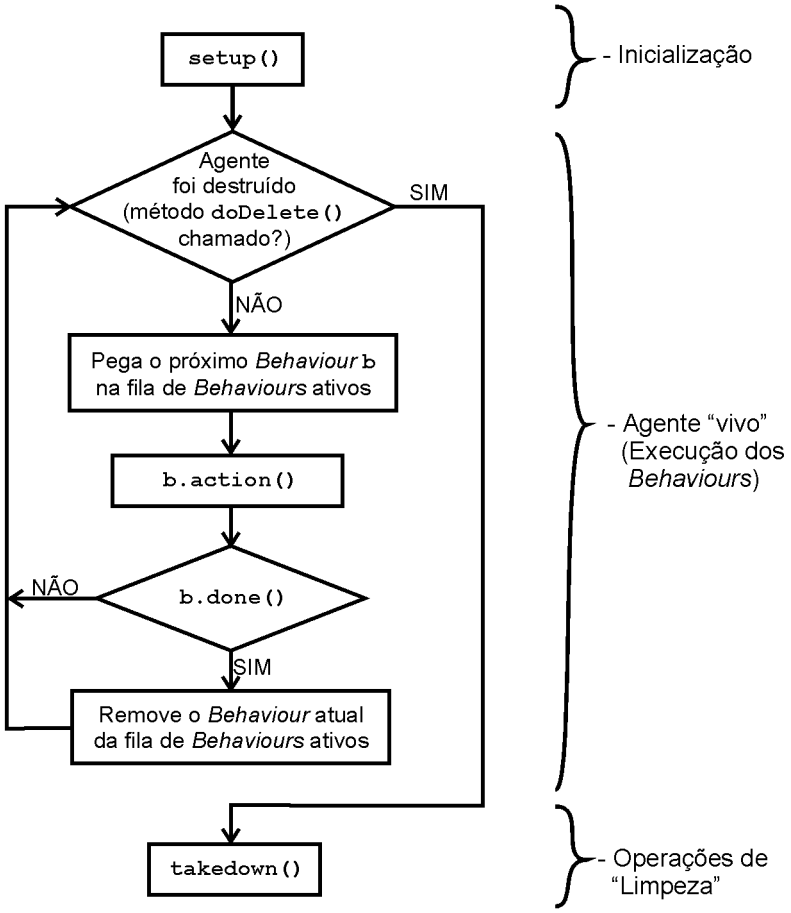



Figura 3.3: Execução interna de um agente JADE

Algumas classes de *Behaviours*, implementadas como classes abstratas, com algum comportamento pré-programado e que podem ser usadas para criar subclasses são:

- `OneShotBehaviour`: para ações que são executadas apenas uma vez; as ações geralmente são simples e de código rapidamente executado.
- `CyclicBehaviour`: para ações executadas ciclicamente, sem um fim especificado; as ações geralmente também são de código rapidamente executado.

- `SequentialBehaviour`: este *Behaviour* contém dentro de si, outros *Behaviours*, sendo que cada sub-*Behaviour* é executado numa seqüência, e termina quando o último dos sub-*Behaviours* terminar.

- `FSMBehaviour`: esta classe também tem sub-*Behaviours*, e estes são executados de acordo com uma máquina de estados finitas definida pelo usuário.

- `WakerBehaviour`: implementa uma tarefa a ser executada apenas uma vez após um determinado período de tempo.

- `TickerBehaviour`: semelhante ao `CyclicBehaviour`, mas cada execução é realizada após um período de tempo determinado.

Além disso, há *Behaviours* para serem usados com alguns protocolos de interação FIPA. Um exemplo é o par de *Behaviours*: `SubscriptionInitiator` e `SubscriptionResponder`, que implementam o protocolo de interação FIPA-Subscribe. A lista completa de Behaviours disponíveis pode ser encontrada atualizada no site do JADE [46].

Um agente em JADE pode estar em diversos estados, de acordo com a especificação FIPA para o ciclo de vida da plataforma de agente (ver figura 3.4, adaptado de [51]):

- Iniciado: o objeto agente foi construído (instanciado), mas não foi registrado com o AMS, não tem nome ou endereço válidos e não pode se comunicar com outros agentes.

- Ativo: o objeto agente está registrado junto ao AMS, tem um nome e endereço válidos e pode acessar todos os recursos do JADE.

- Suspenso: o objeto agente está parado, sua *thread* interna está suspensa e nenhum *Behaviour* está sendo executado.

- Esperando: o objeto agente está bloqueado, esperando por algo; sua *thread* interna está esperando uma condição de sincronização Java e vai retornar a execução quando alguma condição se tornar verdadeira (geralmente, quando uma mensagem chegar).

- Excluído: o agente está definitivamente morto, sua *thread* interna terminou a execução e o agente não está mais registrado no AMS.

- Em trânsito: um agente móvel entra neste estado enquanto está migrando para um novo local; o sistema continua a guardar mensagens num *buffer*, mensagens estas que serão entregues na nova localização [51].

### 3.3.3 Mensagens ACL

O paradigma de comunicação entre os agentes, adotado pelo JADE, é a passagem assíncrona de mensagens. Cada agente tem uma fila de mensagens (estilo *mailbox*). Ao chegar uma mensagem para

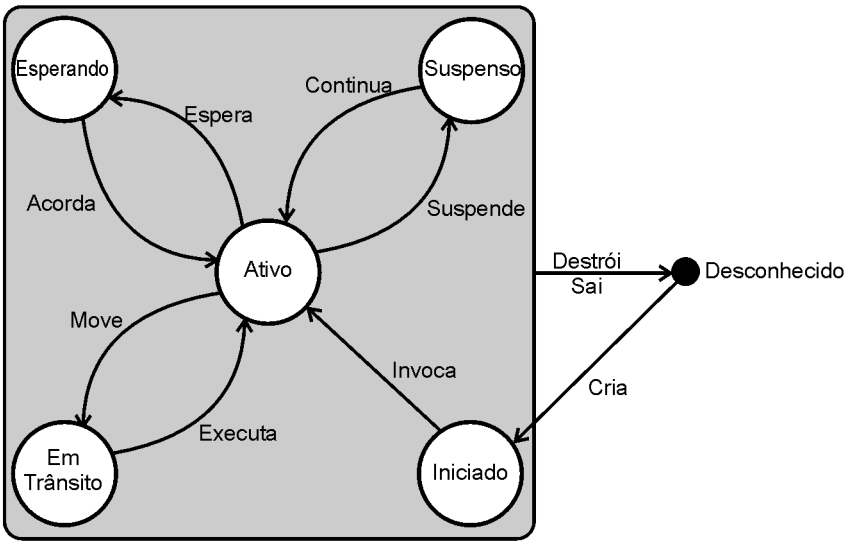



Figura 3.4: Ciclo de vida de um agente definido pela FIPA

um agente, este é notificado, e a manipulação desta mensagem é de responsabilidade de um *Behaviour* (devidamente codificado pelo programador).

As mensagens seguem o formato ACL (Agent Communication Language - Linguagem de Comunicação de Agentes), definida pela FIPA [48]. Dentre os campos de uma mensagem, encontram-se:

- *sender*: o agente que está enviando esta mensagem.
- *receivers*: uma lista dos agentes que devem receber a mensagem.
- *performative*: é a intenção da comunicação. Pode ser: `REQUEST`, se o agente remetente quer que o agente receptor faça alguma ação; `INFORM`, se o remetente quer que o receptor fique a par de um fato; `QUERY_IF`, se o remetente quer saber se determinada proposição é verdadeira ou não; `CFP, PROPOSE, ACCEPT_PROPOSAL, REJECT_PROPOSAL`, usadas quando da negociação entre agentes.
- *content*: o conteúdo propriamente dito da mensagem (por exemplo, se for uma mensagem `QUERY_IF`, o conteúdo é uma proposição em uma linguagem definida, que o remetente deseja saber se é válida ou não para o receptor).
- *language*: indica a linguagem usada para expressar o conteúdo da mensagem.
- *ontology*: indica uma ontologia a ser usada na comunicação.
- *conversation-id, reply-with, in-reply-to, reply-by*: campos usados para controle de várias conversas simultâneas [52].

### 3.3.4 Outros Recursos do JADE

A plataforma JADE tem diversos outros recursos, tais como o suporte a ontologias definidas pelo usuário e suporte a mobilidade de agentes, seguindo a especificação e ontologia de mobilidade de agentes definidas pela FIPA.

Além disso, também implementa outros aspectos como replicação de Containers, tolerância a falhas e outros recursos, disponíveis especialmente para os administradores dos ambientes. Mais detalhes sobre estes recursos em [53].

## 3.4 JESS

JESS (Java Expert System Shell) [54] compreende um shell para Sistemas Especialistas e uma linguagem de script.

JESS foi escrito em Java, o que traz grandes vantagens, tais como: possibilidade de interatividade entre código Java e JESS; possibilidade de se interagir diretamente com elementos JESS através de código Java; além de outras inerentes a plataforma Java.

JESS foi desenvolvido pelo Dr. Friedman-Hill, e atualmente se descreve dando ênfase no motor de inferência (rule engine), uma vez que como pode-se ver em [10], o termo Sistema Especialista se tornou mais amplo do que os Sistemas Baseados em Regras (embora ainda possa se encontrar ambos como sinônimos).

JESS nasceu como um clone do CLIPS [55], portanto o JESS utiliza este estilo de sintaxe. JESS foi implementado utilizando o algoritmo Rete [56], o que o torna extremamente rápido, porém essa velocidade é obtida em troca de consumo de memória. O manual do JESS, também disponível online em [54], traz uma seção onde comenta e analisa a implementação do JESS seguindo o algoritmo Rete.

### 3.4.1 Regras e Fatos

O JESS é um motor de um sistema de regras. Regras são expressões da forma “SE-ENTÃO”.

Em JESS, a parte “SE” é uma lista de proposições que devem ser verdadeiras para a regra poder ser executada. Estas proposições podem ser basicamente de dois tipos: um modelo para um fato constante na base de fatos; ou uma função que deve ser avaliada como verdadeira. Geralmente na construção de um sistema especialista, usa-se mais modelos para fatos. A parte “ENTÃO” é formada por uma lista de funções que (tipicamente) alteram de alguma forma a base de fatos.

Os fatos em JESS podem ser de dois tipos: ordenados e não-ordenados. Fatos ordenados são constituídos simplesmente de uma lista ordenada de elementos, ou átomos, que podem ser símbolos, strings ou números. Abaixo, um exemplo de um fato ordenado, constituído do símbolo “fato”, do número 12 e de uma cadeia de caracteres “um texto qquer”:

```
(fato 12 "um texto qquer")
```

Fatos não-ordenados são muito parecidos com objetos: têm uma "classe" e campos, chamados *slots*, mas com a diferença que não contém métodos. A "classe" dos fatos não-ordenados é chamado de *template*, e é definida com a função `deftemplate`. O exemplo abaixo mostra uma definição de *template*, e um fato deste:

```
(deftemplate humano
    (slot nome)
    (slot idade)
    (slot estado)
)

(humano (nome "Fulano da Silva") (idade 36) (estado casado))
```

Para se definir uma regra, em JESS usa-se a função `defrule`, como ilustrado no exemplo abaixo:

```
(defrule casamento-forcado
    ?fato <- (humano (nome ?n) (idade 21))
=>
   (modify ?fato (estado casado))
   (printout t "O humano "  ?n " acabou de casar")
)
```

Acima, `casamento-forcado` é o nome da regra. A segunda linha faz parte do "SE" da regra e mostra um modelo (ou uma "máscara") para um fato a ser encontrado na base de fatos. Este modelo informa que é um fato de *template* `humano` e com `idade` igual a 21. O `nome` não sofre nenhuma restrição, apenas seu valor é guardado numa variável auxiliar da regra, a variável `?n`. O `estado`, não mencionado, também não sofre nenhuma restrição. O fato em si que combinar com o modelo será armazenado na variável `?fato` (apesar de mais de um fato poder "encaixar" com a máscara fornecida, apenas um fato é escolhido em cada execução da regra).

O símbolo "`=>`" separa as duas partes da regra. Depois dele se encontra a parte "ENTÃO", formada por funções. A primeira função mostrada é a função `modify`, que altera um ou mais slots de um fato não-ordenado. Neste caso, muda o *slot* `estado` de seu valor atual (seja qual for) para `casado`. A segunda função, `printout`, é utilizada para enviar uma mensagem para a tela (ou outra saída). Note que nesta função, usa-se a variável `?n`, definida no modelo para o fato, e que contém o valor do *slot* `nome` do fato encontrado.

Utilizando a regra e *template* definidos acima, apresenta-se uma possível utilização destes no console do JESS. Observe que a função `assert` insere um fato na base de fatos, e a função `run` inicia o motor de inferência:

```
(assert
    (humano
        (nome "Fulano da Silva")
        (idade 36)
        (estado casado))
)
(assert
    (humano
        (nome "Ciclano da Silva")
        (idade 21)
)
(assert
    (humano
        (nome "Beltrano da Silva")
        (idade 21)
        (estado solteiro))
)
(run)

> O humano Ciclano da Silva acabou de casar
> O humano Beltrano da Silva acabou de casar
```

Para maiores informações, a página na Web do JESS [54] contém o material oficial e atualizado, pois novas versões do JESS ainda estão sendo desenvolvidas.

## 3.5 Protegé

Protegé [57] é uma ferramenta para ontologias desenvolvida na Universidade Stanford, utilizando a linguagem Java e seguindo a filosofia open-source.

O modelo de conhecimento do Protegé é compatível com OKBC (Open Knowledge Based Connectivity), incluindo suporte para classes e hierarquia de classes com herança múltipla, *templates* e seus *slots*; especificação de facetas (restrições) arbitrárias e pré-definidas para os *slots*, o que inclui valores permitidos, restrições de cardinalidade, valores padrão e *slots* de valores inversos (contrários); metaclasses e hierarquia de metaclasses.

Do ponto de vista do programador, os principais recursos do Protegé são:

- Modelagem de classes: Protegé fornece uma interface gráfica que modela classes, seus atributos e relacionamentos.
- Edição de instâncias: para as classes modeladas, Protegé automaticamente gera formulários interativos que possibilitam os usuários de entrar com instâncias válidas.

- Processamento do modelo: Protegé tem uma API com extensões que ajudam a definir semântica, executar consultas e definir comportamentos lógicos.
- Troca de modelo: os modelos resultantes (classes e instâncias) podem ser carregados e salvos em vários formatos, incluindo XML, UML, RDF (Resource Description Framework) e até mesmo em banco de dados [58].

Uma das maiores vantagens do Protegé é a sua arquitetura aberta e modular, o que permite aos desenvolvedores adicionar novas funcionalidades criando plug-ins. Há inclusive um plug-in que conecta o Protegé ao JESS (JessTab), possibilitando a definição de regras para derivar novo conhecimento da base de dados existente. Uma lista atualizada dos plug-ins, bem como links e material sobre Protegé pode ser encontrado na sua página Web [57].

## 3.6 Servlets

Servlets, em conjunto com a tecnologia JSP (JavaServer Pages), formam a base sobre a qual as tecnologias Java apresentam interfaces via Web ao usuário.

Interfaces são essenciais em qualquer sistema computacional, pois nenhum sistema computacional é totalmente fechado (se assim o fosse, não teria muita utilidade). E interfaces homem-máquina (IHM) são especialmente importantes, pois muitos sistemas, em última instância, são construídos pelos e para os humanos.

Cada vez mais importantes devido ao crescente aumento da “simbiose” entre humanos e aparelhos, sobretudo sistemas computacionais, as IHM ainda são um campo vasto de estudo, como pode ser visto no capítulo 2 de [59]. Além disso, oferece ainda vastos desafios e dificuldades, bem colocados por Myers em [60].

A expressão Interface Web tem sido usada para designar todas as interfaces que seguem o padrão adotado pela World Wide Web.

O padrão de interface na Web são páginas contendo hipertexto. A linguagem em que este hipertexto é expresso é a HTML (Hyper Text Markup Language) [61], uma linguagem padronizada (pelo World Wide Web Consortium [62], ou W3C, responsável por muitos padrões da Web) que utiliza de marcações dentro do texto, a fim de formatá-lo. De fato, uma página HTML é basicamente texto puro, sendo que um programa navegador (o browser) interpreta as marcações e desenha na tela a interface desejada. A figura 3.5 ilustra este processo, partindo desde a página HTML do servidor Web, até a visão pelo usuário.

No início da Internet, as páginas HTML continham basicamente texto e raramente algumas figuras. Isso era devido tanto ao (pouco) público que acessava a rede (com interesses mais acadêmicos) quanto à limitação de estrutura (baixa banda de rede). Com o crescimento e popularidade da Internet, estas páginas passaram a mostrar mais elementos (como vídeos), mas ainda assim, seus conteúdos continuavam estáticos.

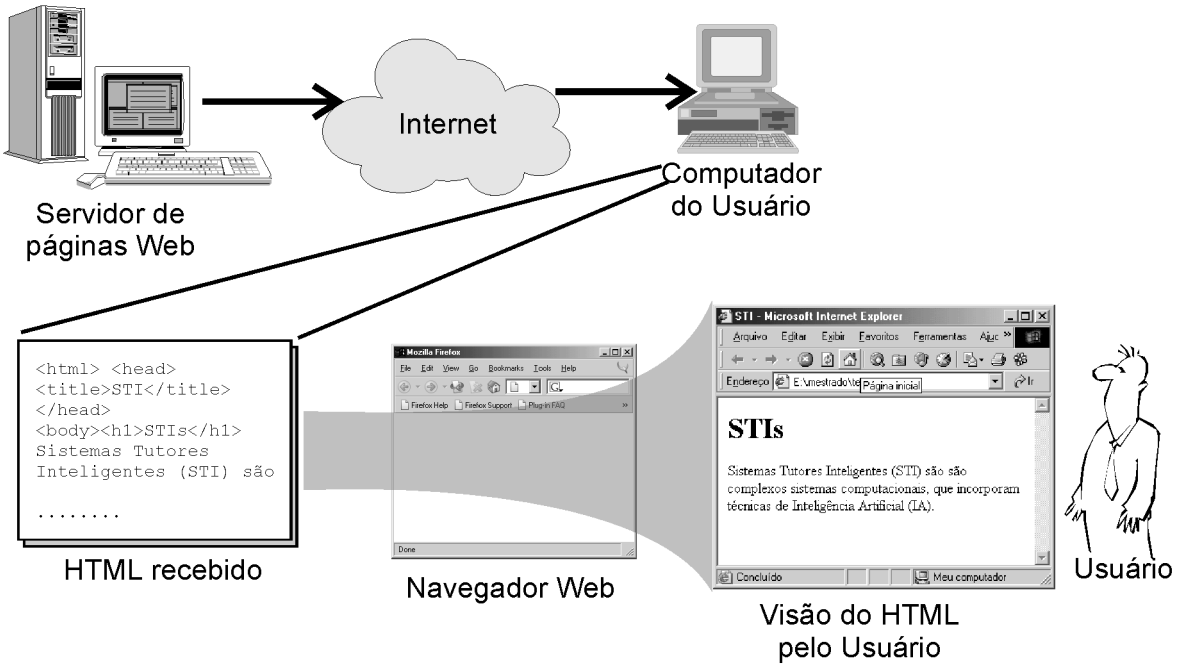



Figura 3.5: O caminho de uma página HTML

Uma forma de interação começou com os formulários (*forms*) HTML. Esses formulários são simplesmente campos dentro de páginas HTML que podem ser preenchidas pelo usuário e cujos conteúdos podem ser enviados para o servidor Web. Além disso, *scripts* também passaram a ser permitidos (e interpretados) dentro do HTML, sobretudo JavaScript.

Estes *scripts* proporcionaram algum incremento na parte cliente (o computador do usuário), sobretudo na parte gráfica, mas o conteúdo em si das páginas continuava estático.

Então, uma nova modalidade de arquitetura surgiu, fazendo com que o conteúdo das páginas não fosse mais estático, retirado de arquivos HTML pré-escritos, mas possibilitando que fosse gerado dinamicamente por programas (figura 3.6). Esta técnica se popularizou devido ao surgimento do CGI (Common Gateway Interface) [63], uma especificação para que servidores de páginas HTML pudessem se comunicar com outros programas. Estes programas, genericamente chamados de *CGI scripts*, no início eram na verdade pequenos programas em C/C++. Atualmente, existe uma diversidade de linguagens sendo usadas, dentre elas as mais conhecidas são Perl, PHP, ASP.Net e JSP.

A tecnologia Servlet [64] é a proposta da tecnologia Java para o desenvolvimento de aplicações estilo CGI. Enquanto nesta última o servidor Web instancia um programa externo (o genericamente chamado script CGI), criando um novo processo no servidor, e desalocando-o da memória quando terminado, no paradigma de Servlets isto não é necessário.

Servlets são programas em Java que residem num servidor, que pode ou não (o mais comum) ser o mesmo servidor responsável pelas páginas HTML estáticas. Este servidor controla o ciclo de vida do Servlet, que fica residente na memória, não necessitando criação/destruição de processos a cada requisição do usuário. O mais conhecido destes servidores é o Tomcat [65].

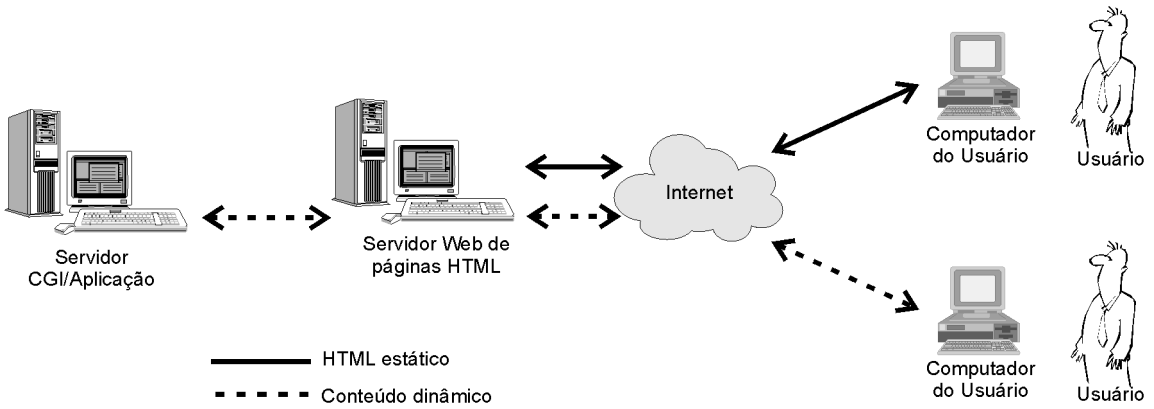



Figura 3.6: Arquiteturas Web

O “trabalho” dos Servlets pode ser resumidos nos seguintes passos, segundo Hall[66]:

- Ler os dados enviados pelo usuário: tipicamente dados vindos de um formulário HTML.
- Recuperar informação da requisição: informação embutida na requisição do protocolo HTTP, tal como qual navegador está sendo usado e outros.
- Gerar resultados: processa os dados e obtém algum resultado.
- Formata o resultado: sendo que o formato final geralmente é uma página HTML a ser visualizada pelo usuário.
- Configura parâmetros de resposta do HTTP: parâmetro como o tipo de documento retornado (HTML, texto puro, entre outros tipos).
- Envia a resposta ao cliente.

Há ainda de se citar os JSP (Java Server Pages) [67], que aparentemente se parecem mais com a linguagem PHP e outros *scripts CGI*, onde o código aparece dentro da página HTML. Entretanto, o próprio servidor que trata dos JSP, os converte para Servlets equivalentes. Abaixo, um exemplo de JSP que exibe a data atual:

```
<HTML>
<HEAD><TITLE>Hello World</TITLE></HEAD>
<BODY>
Hello World, now is <%= new java.util.Date() %>
</BODY>
</HTML>
```

E um Servlet que produz o mesmo resultado:

```
import java.io.*;
import javax.servlet.*;
import javax.servlet.http.*;

public class HelloWorld extends HttpServlet {

    public void doGet(HttpServletRequest request,
                      HttpServletResponse response)
                throws ServletException, IOException {

        response.setContentType("text/html");
        PrintWriter out = response.getWriter();
        out.println("<HTML>\n" +
                    "<HEAD><TITLE>Hello World</TITLE></HEAD>\n" +
                    "<BODY>\n" +
                    "<H1>Hello World, now is "
                         new java.util.Date().toString();
                    "</H1>\n" +
                    "</BODY>\n +
                    "</HTML>");
    }
}
```

Exemplos adaptados de [66].

Devido às suas qualidades e a popularização da própria plataforma Java, Servlets e/ou JSPs têm sido amplamente usados no desenvolvimento de sistemas voltados para Web.

# Capítulo 4

# MathTutor

O sistema MathTutor começou como um Sistema Tutor Inteligente via Web desenvolvido dentro do projeto MathNet [68], destinado a apoiar o ensino de conceitos de abstração de dados e procedimentos, da disciplina de Fundamentos da Estrutura da Informação, do curso de Engenharia de Controle e Automação da Universidade Federal de Santa Catarina (UFSC) [69]. Entretanto, devido à evolução do projeto, atualmente é mais correto identificá-lo como um sistema de autoria de STIs.

## 4.1 O modelo conceitual MATHEMA

A arquitetura do MathTutor é baseada no modelo conceitual MATHEMA [3], um modelo para concepção e desenvolvimento de ambientes de aprendizagem assistidos por computador.

No modelo conceitual MATHEMA o conhecimento sobre um domínio é modelado sob duas dimensões: uma visão externa e uma visão interna.

A visão externa é um esquema de particionamento do conhecimento baseado em suposições epistemológicas, estando comprometido com alguma visão particular do domínio. Consiste de uma perspectiva tridimensional [1]:

- Contexto: compõe-se de diferentes pontos de vista acerca de um domínio de conhecimento constituindo-se de representações ou abordagens diferentes sobre um mesmo objeto de conhecimento;
- Profundidade: relativa a um contexto particular, o conhecimento associado pode ser dividido de acordo com as metodologias usadas para lidar com os seus conteúdos;
- Lateralidade: relativa a um contexto e profundidade específicos, refere-se a conhecimentos complementares que podem ser apontados para permitir que o estudante adquira conhecimentos relacionados ou fora do escopo do curso [70].

Na visão interna, cada conhecimento associado a um sub-domínio é organizado em um ou mais currículos. Cada currículo consiste de um conjunto de unidades pedagógicas (UP) e cada UP é associada a um conjunto de Problemas. Cada Problema contém unidades de interação (UI) que são unidades mais refinadas que apresentam os problemas ao estudante, e que são efetivamente responsáveis pela interação com o estudante.

A arquitetura do modelo MATHEMA consiste de três módulos: uma sociedade de agentes tutores artificiais, uma interface do estudante e uma interface de autoria (figura 4.1, já apresentada na introdução e replicada aqui para melhor visualização).

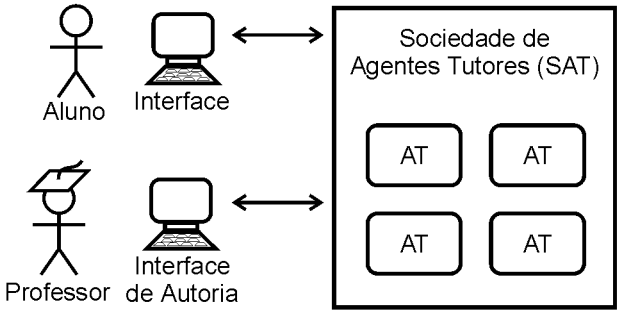



Figura 4.1: Arquitetura do modelo MATHEMA

Na sociedade de agentes tutores artificiais (SATA), cada agente, além das capacidades de comunicação e cooperação, contém um sistema tutor inteligente completo, focado num sub-domínio do conhecimento, modelado de acordo com os parágrafos acima. O fato do sistema consistir de uma sociedade multiagente permite a distribuição dos conteúdos e dados do modelo do aluno entre vários agentes que cooperam no tutoramento [70].

A interface do estudante consiste no STI em si, em que o aluno tem acesso ao tutor. A interface de autoria permite a definição da estrutura do curso e seus conteúdos, ou seja, o conhecimento a ser ensinado.

## 4.2 Modelagem do MathTutor

O Modelo de Domínio adotado pelo MathTutor é uma base de dados com os conteúdos a serem ensinados através de páginas HTML interativas, o que é conseguido através do uso da tecnologia Java Servlets [64] (ver seção 3.6).

O conjunto de páginas HTML é organizado de acordo com grafos de pré-requisitos determinados pelo autor do STI. De acordo com a visão interna do MATHEMA, existe um ou mais currículos, e cada currículo contém Unidades Pedagógicas. O primeiro grafo de pré-requisitos determinado pelo autor é o grafo do currículo, que reflete as relações de pré-requisitos entre as UPs. Para cada UP, é definido outro grafo, que representa as relações de pré-requisitos entre os Problemas daquela UP. Para cada Problema, existe um conjunto de Unidades de Interação, sendo que uma ou mais páginas HTML representam uma UI.

O formalismo adotado para o Modelo do Estudante é um subconjunto da lógica de primeira ordem modelado através de uma ontologia desenvolvida na ferramenta Protegé [57] e suportado pelos mecanismos da base de conhecimento JESS [54]. O modelo contém informação estática, que consiste dos dados de identificação e preferências do estudante, e informação dinâmica, que consiste de descrições das atividades do aprendiz durante todas as seções de interação do aluno com o sistema [2].

Uma vez que a interação geral do estudante com o sistema tutor significa percorrer vários currículos, contidos em vários agentes, os dados acerca das interações do aluno com o sistema ficam distribuídos pelos diversos agentes. Estas informações são basicamente resumos, que contém as UPs visitadas e para cada UP, quais Problemas e Unidades de Interação foram visitadas e o desempenho do aprendiz em cada uma delas.

O Modelo Pedagógico controla a interação entre o aluno e cada agente do sistema. Este modelo é implementado por Redes de Petri Objeto (RPOs), automaticamente traduzidas em sistemas especialistas baseado em regras. As fichas da RPO são compostas por um objeto que identifica o aprendiz. As transições são controladas por condições lógicas que se referem ao Modelo do Estudante e disparam estas transições produzindo ações que atualizam o Modelo do Estudante [2].

De fato, o modelo pedagógico é implementado em dois níveis, cada nível sendo representado por uma RPO.

A Rede de Petri Objeto de primeiro nível, também denominada RPO do Modelo Pedagógico (RPO-MP), reflete a estrutura de relações de pré-requisitos entre as Unidades Pedagógicas. Na RPO-MP os lugares correspondem aos problemas de cada UP, e as transições controlam o fluxo dentro da política de pré-requisitos definida pelo autor do STI. Apesar disso, o aluno ainda pode alterar a sua navegação dentro do curso, através de opções na interface (como por exemplo, revisar conteúdos concluídos, ou então comunicar-se com outros estudantes que estejam também no sistema).

A Rede de Petri Objeto de segundo nível, também denominada RPO do Problema (RPO-Pb), controla a interação do aluno com as várias Unidades de Interação disponíveis para cada problema específico, levando em conta a resposta do estudante e outros atributos do Modelo do Estudante, de acordo com cenários pré-especificados.

## 4.3 Um STI para Fundamentos da Estrutura da Informação

Como exemplo ilustrando as sessões anteriores, é mostrado a seguir a modelagem do STI desenvolvido para a disciplina Fundamentos da Estrutura da Informação.

Seguindo o modelo MATHEMA, na visão externa o domínio de conhecimento é dividido em dois contextos: teórico e prático; e cada contexto é trabalhado em duas profundidades: abstração procedural e abstração de dados. Desta forma, a sociedade multiagente (SATA) consiste de quatro agentes tutores, cada um responsável por um dos seguintes sub-domínios [2]:

- PT - Abstração procedural teórica;
- PP - Abstração procedural prática;
- DT - Abstração de dados teórica;
- DP - Abstração de dados prática.

Neste modelo, as UPs para cada contexto (teórico e prático) são os mesmos, apenas a abordagem utilizada dentro de cada UP é diferente.

Para ilustrar, um possível currículo com foco na abstração procedural (PT ou PP) é mostrado na figura 4.2 abaixo:

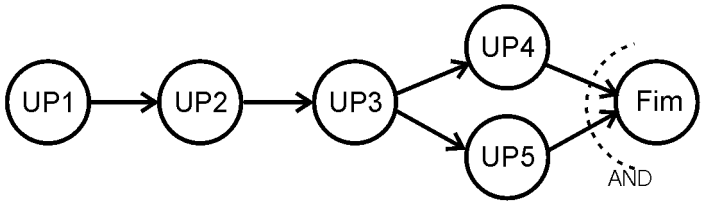



Figura 4.2: Grafo de Pré-Requisitos para um currículo do STI de Fundamentos da Estrutura da Informação

Neste currículo, as Unidades Pedagógicas, com seus respectivos problemas são:

- UP1 - Introdução
- UP2 - Elementos de Programação

  Pb1 - Expressões

  Pb2 - Tipos primitivos de dados

  Pb3 - Expressões condicionais

  Pb4 - Funções
- UP3 - Processos e Procedimentos

  Pb1 - Funções Recursivas

  Pb2 - Funções Iterativas

  Pb3 - Ordem de Crescimento
- UP4 - Funções de Ordem Superior

  Pb1 - Funções como Argumento

  Pb2 - Uso do Lambda
- UP5 - Abstração de Dados

  Pb1 - Listas

  Pb2 - Manipulação de Listas

Note que apesar de uma UP ter como conteúdo a abstração de dados (UP5), o foco dos conteúdos são os procedimentos.

Na figura 4.3 é mostrado o grafo com as UPs expandidas, visualizando os problemas dentro de cada UP.

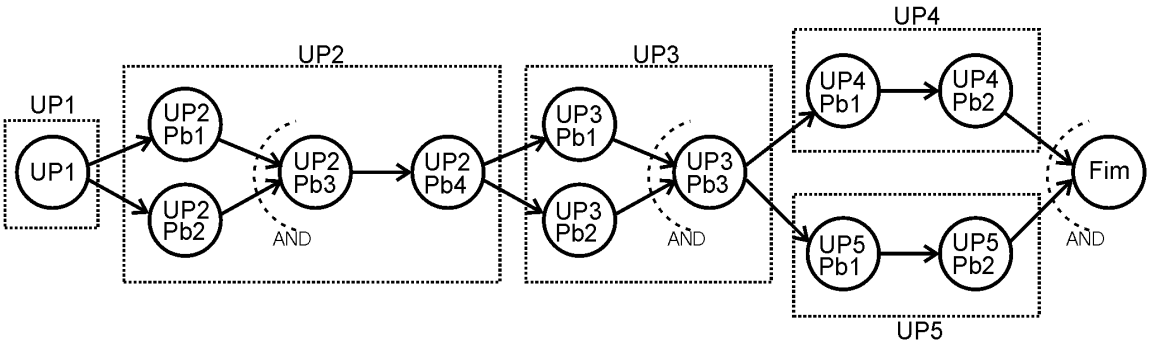



Figura 4.3: O Grafo de Pré-Requisitos com os Problemas visualizados

Do grafo mostrado na figura 4.3, é gerada a RPO-MP. Para cada problema representado na RPO-MP podem existir diversos RPO-Pb, cada uma com uma abordagem pedagógica diferente. O comportamento das RPO-Pb são projetados pelos desenvolvedores do MathTutor, restando ao professor apenas fornecer os conteúdos de acordo com a definição da rede.

Nesta implementação, apresenta-se como exemplo uma RPO-Pb bastante simples, contendo apenas UI simples (páginas HTML sem grande interatividade). Ver figura 4.4.

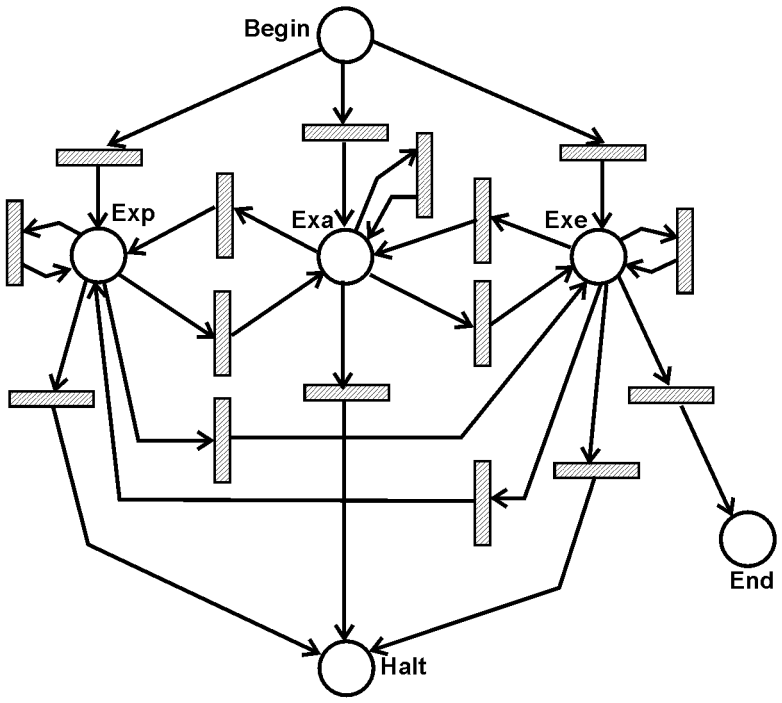



Figura 4.4: Rede de Petri Objeto de segundo nível do Modelo Pedagógico

Os lugares Exp, Exa e Exe estão associados, respectivamente, às UIs de explicação, exemplo e exercício, ou seja, quando uma ficha se encontrar num destes lugares, a UI correspondente será ativada e apresentada ao aluno. Apesar do tipo da UI ser sempre o mesmo, a atividade em si é diferente a cada vez que uma ficha se encontrar em cada lugar, pois durante a construção do curso o autor do STI associa diversas opções de explicações, exemplos e exercícios para cada problema, e o sistema identifica quais opções já foram vistas, pois isso consta da parte dinâmica do Modelo do Estudante.

Os demais lugares da RPO-Pb são lugares de controle. O lugar Begin corresponde a uma apresentação central do problema, que leva uma das UIs dependendo do Modelo do Estudante ou o seu *feedback*. O lugar Halt sinaliza ao sistema que por algum motivo, o estudante decidiu terminar a interação com o sistema. O lugar End sinaliza ao sistema que o aluno concluiu este problema da UP.

# Capítulo 5

# Implementação

Como visto na seção 4.1, dentre os módulos do MathTutor (ver arquitetura na figura 4.1), há uma sociedade de agentes tutores artificiais (SATA), sendo que cada um destes agentes contém dentro de si, um sistema tutor completo. Este módulo se comunica com o módulo da Interface do Estudante, como mostra a figura 5.1.

Neste capítulo abordaremos os módulos computacionais implementados.

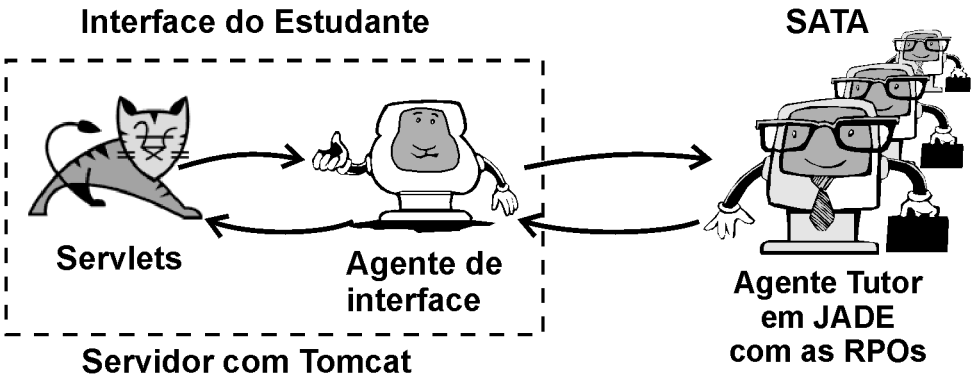



Figura 5.1: Elementos Implementados

Na figura 5.1 pode-se ver que alguns módulos residem dentro do servidor Tomcat. Estes fazem parte da Interface do Estudante, e são o elo de ligação entre o aluno e os agentes tutores da SATA. O módulo dos Servlets consiste num conjunto de Servlets implementados, especializados na manipulação da interface Web com o estudante. Estes, por sua vez, se comunicam com um agente de interface. O agente de interface implementado consiste de um agente que faz a ligação entre os Servlets e a sociedade dos agentes tutores. Este agente é necessário pois os Servlets não conseguem diretamente lidar com a troca de mensagens entre os agentes, que é o seu meio de comunicação.

Além disso, um tipo de agente tutor também foi implementado. Trata-se de um agente tutor genérico, ou seja, o tutorial/curso que ele contém dentro de si não é estático. De fato, o tutor que ele carrega depende da definição de um curso provido por um instrutor/professor, o que deverá ser feito pelo módulo da Interface de Autoria.

Além disso, dois módulos que compõem parte da Interface de Autoria (ver 4.1) do modelo do MathTutor foram implementados. Estes módulos fazem parte do fluxo que a definição de um curso deve seguir, e estão representados por desenhos de computadores na figura 5.2.

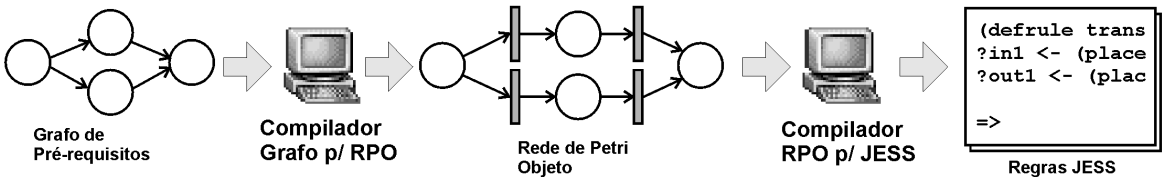



Figura 5.2: Fluxo da definição de um curso - Interface de Autoria

Com isso, pode-se perceber que cada módulo do modelo MATHEMA (ver seção 4.1) ilustrado na figura 4.1 teve sua estrutura computacional básica implementada.

A seguir, analisaremos alguns dos módulos implementados, em maior profundidade.

## 5.1 Um agente tutor em JADE

A SATA contém diversos agentes tutores, cada qual, contendo a definição de um curso completo. Cada agente tutor pode, além da definição de curso diferente, ser implementado de maneira diferente. Neste trabalho, apresentamos apenas um tipo de implementação de agente.

A classe do agente tutor no presente trabalho é `StudentPetriNetsAgent`, e contém dentro de si, duas instâncias do *engine* JESS, cada uma contendo uma das RPOs, como mostrado em 4.2.

Agentes em JADE executam as suas ações por meio de seus *Behaviours* (comportamentos). A interação entre o agente e cada uma das RPOs em JESS é feita via os *Behaviours* especializados, pois as interações são ações do agente. Para cada nível da RPO, um *Behaviour* é implementado. Cada um destes recebe requisições vindas do JESS (o envio de informações/dados citados na seção anterior), e lida com estas requisições. Por exemplo, no caso do *Behaviour* que lida com a RPO de 1º nível, ao receber uma requisição para enviar uma ficha à RPO de 2º nível, insere no JESS da RPO de 2º nível uma nova ficha representando a ficha da RPO de 1º nível.

O diagrama de classes mostrando o agente e seus *Behaviours* é mostrado na figura 5.3.

As classes `OpnFirstLevelMessageBehaviour` e `OpnSecondLevelMessageBehaviour`, ambas subclasses de `OpnInternalMessageBehaviour`, lidam com as requisições vindas do JESS, relacionadas às RPOs de primeiro e segundo níveis, respectivamente (essas requisições podem ser descritas como mensagens na terminologia da UML, daí a denominação das classes). A classe `InterfaceInitiator`, *inner class* de `OpnSecondLevelMessageBehaviour`, lida com a interação com outro agente (por meio do protocolo de interação FIPA-Request [48]), um agente de interface com o usuário para a qual a requisição da RPO de 2º nível é encaminhada.

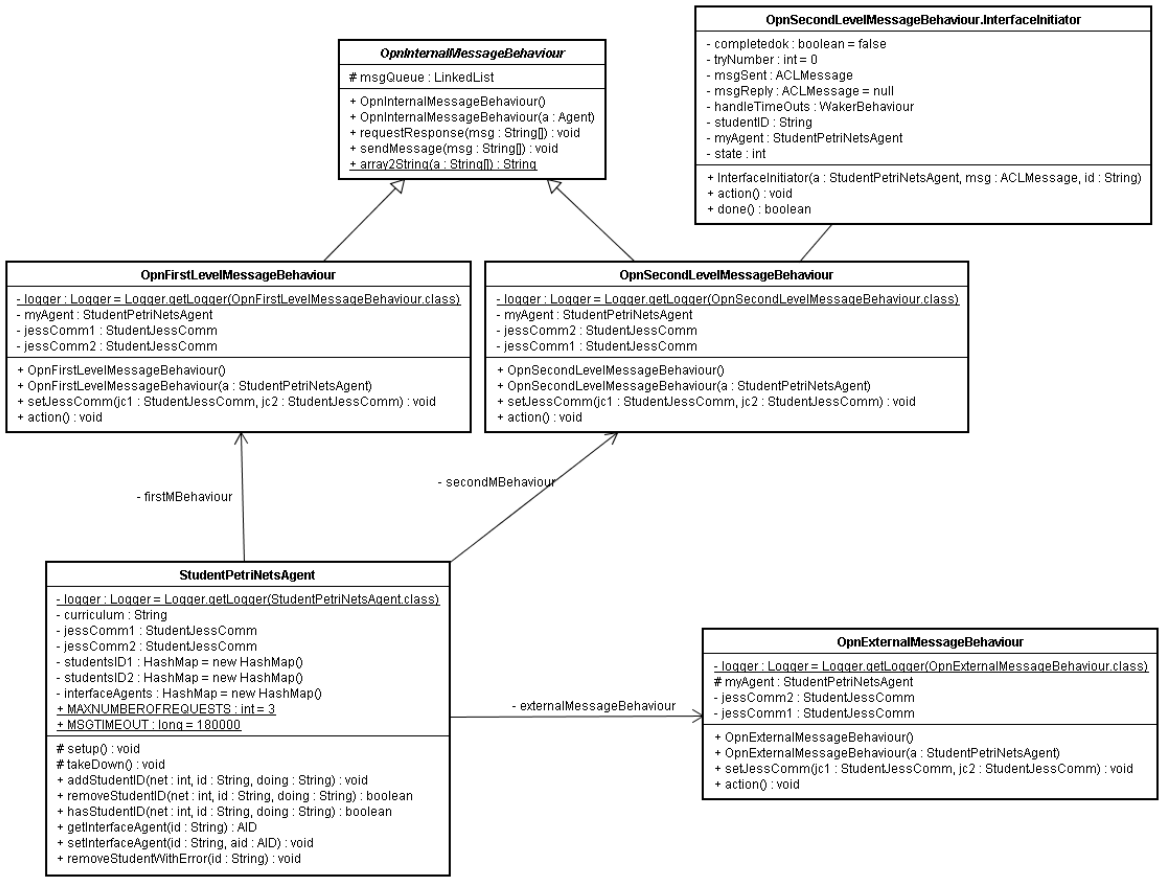



Figura 5.3: Diagrama de classes do agente StudentPetriNetsAgent

A classe `OpnExternalMessageBehaviour` lida com mensagens vindas de outros agentes, tal como mensagens requisitando ingresso de um aluno no tutor (significando a inserção de uma ficha correspondente ao estudante na RPO de 1º nível).

Uma descrição mais detalhada (e pragmática do ponto de vista do programador) da implementação deste agente se encontra no apêndice D, bem como diagramas mais fáceis de serem visualizados, pois estão em mais de uma página (devido ao tamanho).

## 5.2 Um compilador para transformar uma Rede de Petri Objeto em um sistema de regras

Como já citado na seção 4.2, o modelo pedagógico é implementado utilizando-se duas Redes de Petri Objeto (RPO).

Redes de Petri Objeto são ferramentas matemáticas utilizadas freqüentemente para modelagem de sistemas. Entretanto, para a construção do Sistema Tutor Inteligente, deve-se utilizar uma ferramenta que “execute” o modelo em RPO. As ferramentas existentes são em sua grande maioria focadas na modelagem, análise e simulação de Redes de Petri, sobretudo as ordinárias. Para execução de uma RPO, as ferramentas são mais escassas, segundo guia encontrado em [71].

Para executar as RPOs, optou-se por utilizar um *shell* de Sistemas Especialistas baseado em regras, regras estas que implementam uma RPO. Além disso, o uso de um sistema de regras traz a possibilidade de se adicionar novas capacidades/habilidades, por meio de regras específicas (não relacionadas diretamente com a RPO). A opção de tal *shell* recaiu sobre o JESS [54], devido sobretudo a sua forte integração com Java (uma vez que ele é implementado nesta linguagem).

Primeiramente, definiu-se a estrutura da RPO na base de fatos e regras do JESS: cada lugar da rede corresponde a um fato do *template* (tipo de fato em JESS) lugar; cada transição da rede corresponde a um fato do *template* de um tipo de transição (por tipo de transição, entenda-se quantos lugares de entrada e quantos lugares de saída uma transição possui); os arcos são implementados como propriedades dos fatos de transição; as condições de disparo, bem como as ações são funções ligadas a propriedades dos fatos de transição.

Em seguida, utilizando-se a ferramenta JavaCC [42], foi construída uma classe de *parser* (análise sintática) para a gramática que descreve uma RPO (gramática proprietária - ver no apêndice A).

Em conjunto com o *parser*, um conjunto de classes representando as estruturas da RPO foram desenvolvidas. Estas classes contém métodos que geram códigos em JESS que implementam a RPO. O diagrama de classes com alguns comentários se encontra no apêndice B, bem como diagramas e uma descrição mais detalhada de todo o código.

## 5.3 Outras regras: adicionando semântica complementar na RPO

Apesar do compilador descrito na seção anterior gerar as regras necessárias para uma Rede de Petri Objeto ser executada, o compilador não gera automaticamente algo: a semântica associada a cada lugar na rede. Até mesmo por razões de portabilidade (uma vez que cada RPO pode ter a sua própria semântica relacionada a um lugar), a semântica é inserida em um arquivo em separado, arquivo este que contém o código da semântica já implementado em JESS.

Na RPO de 1º nível, cada lugar representa um problema de uma Unidade Pedagógica a ser feito pelo Estudante. Entretanto, este problema desmembra-se nas Unidades de Interação (explicação, exemplo e exercício), que é de domínio da RPO de 2º nível. Portanto, a semântica de um lugar na RPO de 1º nível é encaminhar o Estudante para as unidades de interação da RPO de 2º nível, e esperar o seu término ali. Abaixo, uma regra que implementa esta semântica, descrita em linguagem natural:

```
SE:
- A ficha F está no lugar L
- E o estudante que é representado pela ficha F ainda não está
  executando nenhum IU
ENTÃO:
- Envia a ficha F para a RPO de 2º nível
```

Na RPO de 2º nível, os lugares ou apresentam uma Unidade de Interação (UI), ou um lugar de semântica de controle. Os lugares de UI são: Explanation (Explicação), Example (Exemplo) e Exercise (Exercício), e os de controle são Begin, Halt e End (ver figura 4.4). Exceto os lugares de controle Halt e End, a regra que implementa a semântica dos lugares de UI é:

```
SE:
- A ficha F está no lugar L
- E o estudante representado pela ficha F ainda não deu uma resposta para a UI
- E a ficha não está em modo de espera pela resposta
ENTÃO:
- Envia os dados da UI para a interface com o usuário para esperar resposta
```

Os lugares Halt e End têm a regra um pouco diferente: ao invés de enviar os dados da UI, enviam um comando informando que a interação com o usuário terminou, enviando também dados do estudante para persistência.

Nas regras de ambas as RPOs, a parte "Então" da regra exige o envio de alguma informação para fora da RPO, sendo a própria ficha da rede para a rede de segundo nível, no caso da RPO de 1º nível, ou então dados de alguma Unidade de Interação para o usuário, no caso da RPO de 2º nível. Na implementação concreta das regras, este envio exige comunicação do código em JESS para fora do escopo deste. Para isto, foi desenvolvido um conjunto de classes que implementam boa parte da lógica necessária para controle dessa comunicação.

No sistema implementado, a comunicação se dá de maneira assíncrona, e o processo é mostrado detalhadamente no apêndice C.

## 5.4 Interface com o Estudante: um agente em JADE e integração com Servlets

A interface com o aluno se faz via Web, através de páginas dinâmicas implementadas com Servlets. Portanto, há a necessidade de se fazer com que o sistema multiagente e os Servlets se comuniquem. A solução adotada foi a de implementar um agente, denominado de Agente de Interface, que reside não no mesmo Container que os agentes tutores da SATA, mas num Container diferente, pertencente ao processo do servidor Tomcat [65]. Esta abordagem foi escolhido pois o Agente de Interface precisa ter forte vínculo com os Servlets, e ao mesmo tempo, a comunicação com a sociedade de Agentes Tutores não é prejudicada por estarem em diferentes Containers (o que equivale dizer estarem em diferentes máquinas virtuais Java, ou diferentes processos no Sistema Operacional). Ver figura 5.4.

Ao entrar no sistema (fazendo *login* numa página Web, passando por um Servlet), é criado dentro do Tomcat uma instância da classe `ServletStudentAgent`, que representa o usuário no sistema multiagente. Esta classe contém referência a um *Container* único do Tomcat, que é criado dinamicamente caso ainda não exista, e se conecta a outro *Container*, principal, no qual residem os Agentes Tutores.

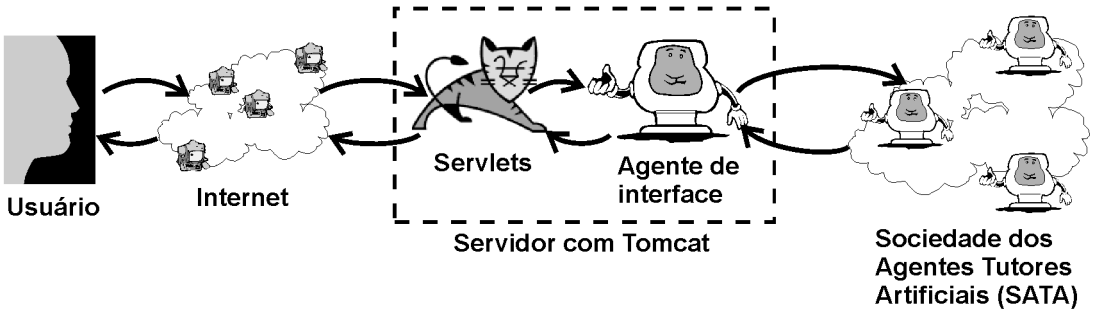



Figura 5.4: Arquitetura da interface Web

A interação entre Servlets e o agente é feita passando-se um objeto da classe StudentInteraction para o agente. Esta classe contém métodos sincronizados, que assim possibilitam o agente de "responder" requisições do Servlet (pois apesar de ser permitido enviar para o agente objetos representando mensagens, oriundas de classes não-agentes Java, não existe mecanismo similar para receber do agente este mesmo tipo de objetos). Este padrão está ilustrado no diagrama de seqüência da figura 5.5.

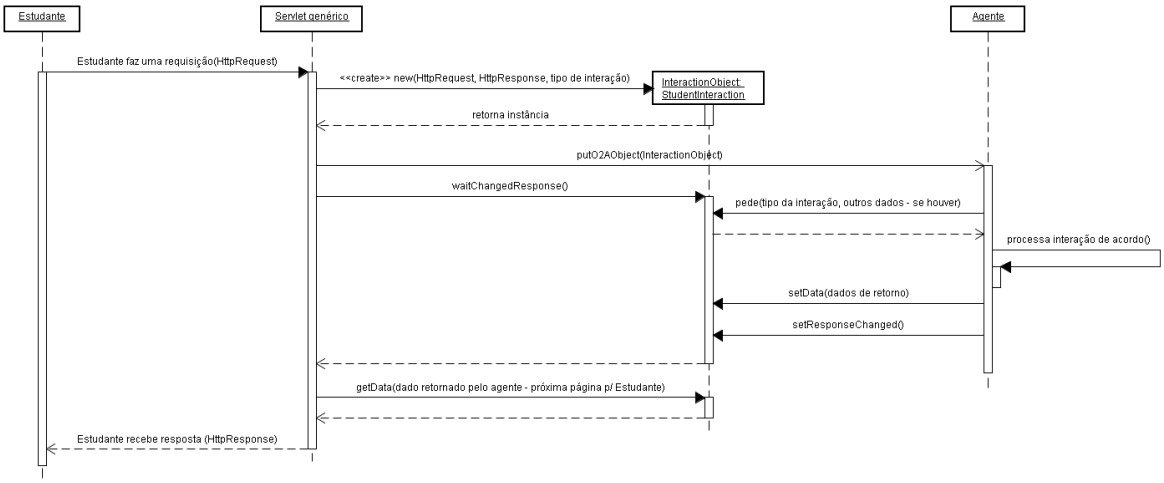



Figura 5.5: Diagrama de Seqüência do padrão de interações entre Usuário/Servlets/Agente

O agente da classe `ServletStudentAgent` tem basicamente quatro *Behaviours*:

- `GetInfoBehaviour`: procura as configurações do sistema multiagente;
- `Servlet2AgentHandlerBehaviour`: lida com as requisições vindas dos Servlets;
- `PetriNetInitiator`: lida com o protocolo FIPA-Request quando da requisição de inserção de um estudante num Agente Tutor (ou seja, da inserção de uma ficha na RPO de 1º nível daquele agente);
- `PetriNetResponder`: lida com as requisições vindas do Agente Tutor, geralmente quando a RPO de 2º nível está requisitando uma resposta.

A interação entre um agente ServletStudentAgent e um Agente Tutor (descrito na seção anterior) segue o protocolo de interação FIPA-Request, e é mostrado simplificadamente na figura 5.6.

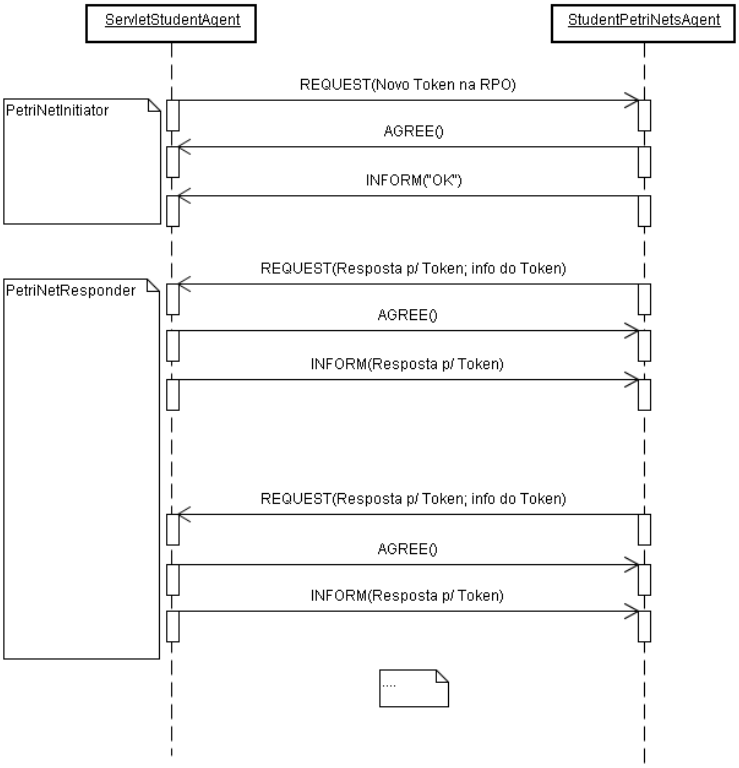



Figura 5.6: Diagrama de Seqüência ilustrando interação entre dois agentes

Note-se que todas estas comunicações/interações entre os agentes, bem como internamente ao agente (no caso do agente tutor), seguem uma ontologia em desenvolvimento. Esta ontologia, desenvolvida com a ferramenta Protegé [57], ainda não está completa. A integração da ferramenta Protegé com as demais ferramentas usadas também não foi completada, portanto não é aprofundada a discussão deste aspecto do sistema.

Uma descrição mais detalhada da implementação deste agente, bem como descrições do código dos Servlets e diagramas mais fáceis de serem visualizados se encontram no apêndice E.

# Capítulo 6

# Conclusões

Procurou-se apresentar neste trabalho uma breve discussão sobre Sistemas Tutores Inteligentes e arcabouços para o desenvolvimento/construção destes, área de pesquisa na qual se encontra o projeto MathTutor, baseado no modelo conceitual MATHEMA, ambos também discutidos ao longo do texto.

Mostramos a concepção, modelagem e implementação de um agente tutor pertencente à sociedade de agentes tutores da arquitetura do MathTutor, bem como sua estrutura de controle interna, construída com o motor de inferência de Sistemas Especialistas JESS, com regras emulando Redes de Petri Objeto. Mostramos também como as RPOs modelam o currículo de um curso/tutorial, bem como o seu modelo pedagógico, e sobretudo mostramos a passagem do modelo teórico para um caso prático, no caso a implementação mostrada, que serve até mesmo com prova de conceito.

Também mostramos regras adicionais, que não fazem parte de uma RPO, mas que são necessárias para a comunicação entre o nível da RPO e o estudante, passando pelas estruturas dos agentes tutores em JADE.

Como suporte ao agente da SATA, foi também desenvolvido o módulo da interface do estudante, baseado em páginas Web dinâmicas, através do uso de Servlets, integradas ao sistema multiagente através de outro agente especialmente desenvolvido para tal.

## 6.1 Trabalhos Futuros

Vários trabalhos complementares ou derivados ainda estão por fazer. Um exemplo é a integração de todo o sistema com a ontologia desenvolvida com o Protegé.

Além disso, a interface de autoria do STI, apesar de descrita, ainda precisa ter elementos implementados e integrados com o resto do sistema em funcionamento.

A colocação em funcionamento efetiva de um STI para ser usado e avaliado por estudantes também é o próximo passo. Em função da avaliação e *feedback* dos alunos, pode-se ter uma boa indicação de novos rumos e melhoramentos a serem tomados.

# Apêndice A

# Gramática da descrição da Rede de Petri Objeto

Como visto na seção 4.2, o modelo pedagógico do MathTutor é modelado usando Redes de Petri Objeto. Para expressar a RPO foi desenvolvida uma gramática particular, que serve de entrada para o compilador que transformará a RPO num sistema de regras, como visto na seção 5.2. Esta gramática é apresentada a seguir:

```
<petri-net> →

        PETRINET
          <def-classes>
          <def-places>
          [<def-objects>]
          <def-transitions>
          [<def-conditions>]
          [<def-actions>]
          [<def-functions-file>]
        ENDNET

<def-classes> →

 Class := {
    [tokenclass :] <class-name>:
    { ( [multifield] <attribute-name> [<attribute-value>] ) }+
    ;
  }+
```

```
<def-places> →

 Places := {
    <place-name>:
    (<class-name> {, <class-name}*)
    ;
  }+

<def-objects> →

 Objects := {
    <class-name>.<object-name>:
    { (<attribute-name> <attribute-value>) }*
    ;
  }+

<def-transitions> →

 Structure := {
   <transition-name>:
     ( <place-name> ( <class-name> {, <class-name>}* )
       {, <place-name> ( <class-name> {, <class-name>}* )}*
     )
     { -> | => }
     ( <place-name> ( <class-name> {, <class-name>}* )
       {, <place-name> ( <class-name> {, <class-name>}* )}*
     )
     ;
   }+

<def-conditions> →

 Conditions := {
   <transition-name>: <function-name> ( <parameters> )
   ;
 }+
```

```
<def-actions> →

 Actions := {
   <transition-name>: <class-name>.<attribute-name> :=
     <function-name> ( <parameters> )
      |
     <attribute-value>
   {
     , <class-name>.<attribute-name> :=
        <function-name> ( <parameters> )
         |
        <attribute-value>
   }*
   ;
 }+

<parameters> →

 [
   <identifier> | <number> | <string>
    |
   <object-name>.attribute.name>
    |
   <class-name>.attribute.name>
    |
   <function-name> ( <parameters> )
   { ,
     <identifier> | <number> | <string>
      |
     <object-name>.attribute.name>
      |
     <class-name>.attribute.name>
      |
     <function-name> ( <parameters> )
   }*
 ]
```

<def-functions-file> ⟶

FunctionFile := <string>

<identifier> ⟶

{ {a-z} | {A-Z} | _} { {a-z}{A-Z}{0-9}{_} }*

<number> ⟶

{0-9}{0-9}*

<string> ⟶

" { {a-z}{A-Z}{0-9}{-}{_}{.}{ } }* "

<attribute-value> ⟶

<identifier> | <string> | <number>

# Apêndice B

# Transformando uma Rede de Petri Objeto em um Sistema de Regras

Como visto na seção 4.2, o modelo pedagógico do MathTutor é modelado usando Redes de Petri Objeto. Para poder "executar" a RPO, como já dito na seção 5.2, é utilizado um sistema de regras, mais especificamente o JESS. Como "ponte" entre o modelo de Redes de Petri Objeto e o sistema de regras em JESS, foi desenvolvido um compilador, cuja finalidade é transformar a RPO numa representação apropriada para ser usada no JESS (ver figura 5.2).

Nesta seção é mostrado detalhadamente, e passo a passo, o processo de construção deste compilador.

## B.1 Primeiro Passo - Análise Sintática

O primeiro passo é fazer a análise sintática (*parsing*) de um arquivo contendo uma representação da RPO, conforme padrão estabelecido (ver apêndice A).

Para executar o parsing, foi usada a ferramenta JavaCC (ver seção 3.2).

O JavaCC é uma ferramenta que gera um programa (na verdade, uma classe) em Java, que analisa sintaticamente uma determinada construção, de uma gramática.

O JavaCC permite expressar uma Gramática Livre de Contexto quase sem modificações, apenas substituindo as variáveis/produções por métodos, como é mostrado no exemplo abaixo, retirado da gramática de construção da RPO:

```
<petri-net> ::= PETRINET
                <def-classes>
...

<def-classes> ::=
    Class :=
...
```

Expressando em JavaCC:

```
void PetriNet() throws Exception : { }
{
    <PETRINET>
    DefClasses()
...

void DefClasses() throws Exception : { }
{
    <CLASS>
    ":="
...
```

onde:

- `<PETRINET>` e `<CLASS>` são *tokens* especiais, que levam a literais, ou seja, palavras reservadas;
- { } é um espaço reservado para a declaração de código em Java;
- `throws Exception` indica que em caso de algum erro, uma exceção seja enviada para o método que iniciou o *parsing*.

Outro exemplo, mais simples - considere a gramática:

S → A — B

A → aB

B → b

Expressando na forma do JavaCC:

```
void S() : {}
{
```

```
   A() | B()
}

void A() : {}
{
   "a" B()
}

void B() : {}
{
   "b"
}
```

Entretanto, somente a descrição da gramática não é o suficiente para o JavaCC funcionar. São requeridos outros detalhes, que podem ser encontrados na documentação do JavaCC [42] (mais precisamente nos seus exemplos).

## B.2 Segundo Passo - Adicionar Semântica na Gramática

O segundo passo é instanciar estruturas de dados representando a RPO.

Somente com a descrição da gramática, o JavaCC não gera resultados muito interessantes: a classe só consegue apontar erros quando, na entrada, os dados não estão conforme a gramática.

Portanto, é preciso inserir alguma semântica dentro das produções da gramática. Isso é feito instanciando/manipulando algumas estruturas de dados. Por exemplo, na produção de declaração de classes:

```
[tokenclass :] <class-name> :
{
   ( [multifield] <attribute-name> [<attribute-value>] )
}+
;
...
```

Em JavaCC:

```
[ token_class = <TOKENCLASS> ":" ]
name = <ID>
{
  if (token_class != null)
```

```
    cl = new OpnClass(name.toString(), null, true);
  else
    cl = new OpnClass(name.toString(), null, false);
}
":"

(
  "("
  [ multi = <MULTIFIELD> ]
  attr = <ID>   [ ( attr_value = <ID> ) |
                  ( attr_value = <STRING> ) |
                  ( attr_value = <NUM> ) ]
  ")"
  {
    if (multi == null)
      if (attr_value == null)
        cl.addField(attr.toString(), null);
      else
        cl.addField(attr.toString(), attr_value.toString());
    else
      if (attr_value == null)
        cl.addMultiField(attr.toString(), null);
      else
        cl.addMultiField(attr.toString(), attr_value.toString());
    multi = null;
    attr_value = null;
  }
)+

";"
```

No primeiro bloco de código, temos a criação de um objeto (`cl`) da classe `OpnClass`. No segundo bloco, temos a manipulação deste objeto, via as chamadas de alguns de seus métodos. O diagrama de classes contendo as estruturas usadas no compilador está na figura B.1.

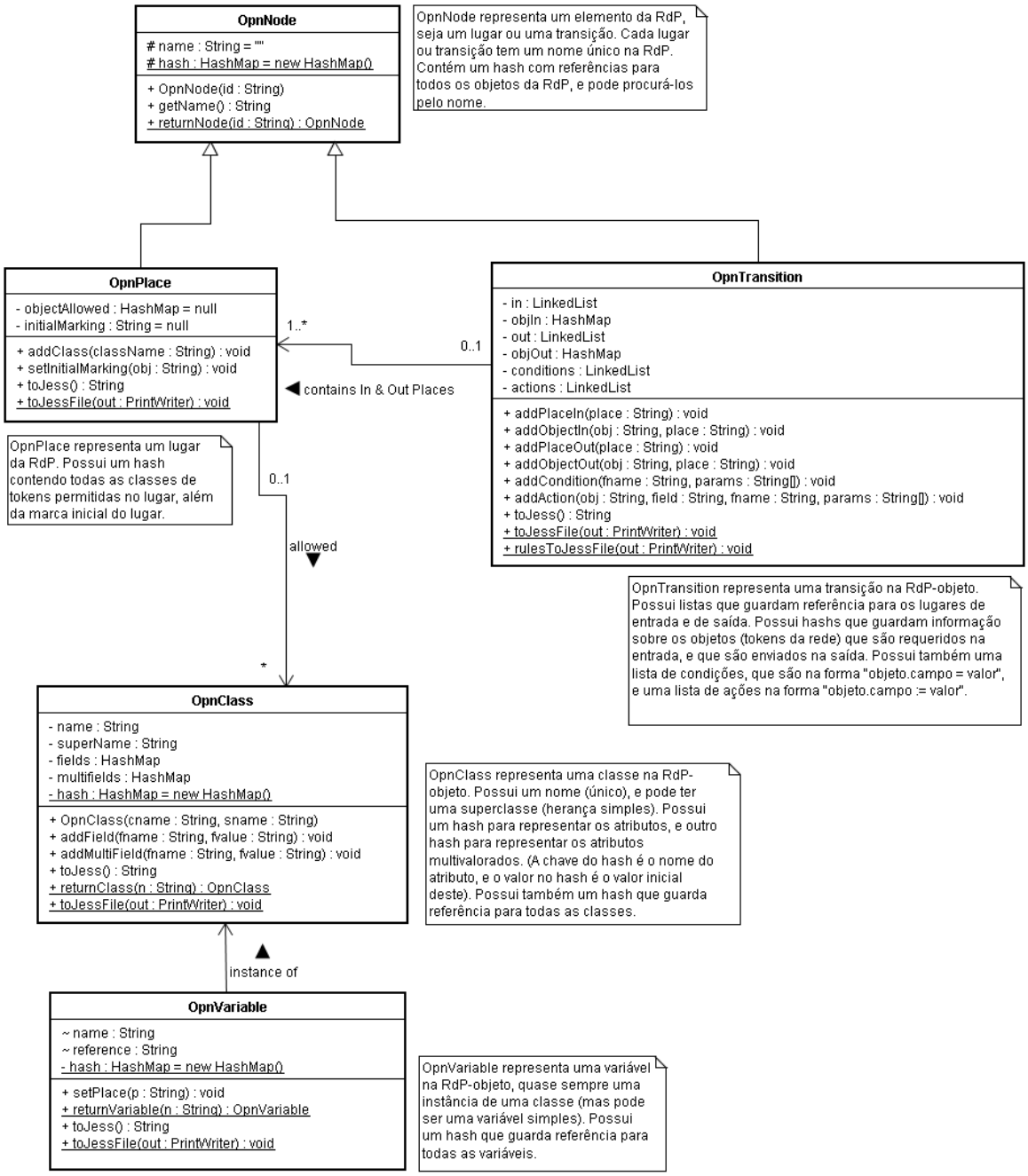



Figura B.1: Diagrama de Classes do compilador RPO/JESS

## B.3 Terceiro Passo - Implementar as Estruturas de Dados

O terceiro passo é implementar nas estruturas de dados, métodos que transformem a RPO num sistema de regras.

Executado o parsing, e instanciadas as estruturas de dados sem erros, o próximo passo é "traduzir" estas estruturas em forma de um sistema de regras. Para tal, vamos analisar cada uma das estruturas de dados separadamente:

### B.3.1 OpnClass

`OpnClass` representa uma classe na RPO. Pode ou não ser uma ficha na rede.

É nesta classe que estão definidos os campos que os objetos terão. Esses campos podem ser campos simples ou multicampos (contém uma lista de quantidade arbitrária de valores), e podem ou não terem valores padrão.

No sistema de regras JESS, a classe é representada por um *template*, cujo nome é o nome da classe, contendo os campos e multicampos definidos.

Um detalhe: classes que são fichas da RPO, contém dois campos padrão: um multicampo `where` (que indica em quais lugares a ficha se encontra) e um campo `busy` (que está inserido ai por razões semânticas, indicando que uma ficha está ocupada, e as transições da rede não podem ser disparadas com ela).

### B.3.2 OpnPlace

`OpnPlace` representa um lugar na RPO. Um `OpnPlace` possui um nome único, e tem como propriedades, além do nome:

- Uma designação de tipo de lugar (que por enquanto, possui apenas dois valores: `problem` - valor padrão - e `buffer`). Para a identificação do tipo de lugar, é usado uma convenção: se o nome do lugar começar com "buf", então o tipo será `buffer`.
- Um conjunto de classes (fichas na RPO) que podem ocupá-lo. Este conjunto é usado durante o *parsing* da RPO, para averiguação da corretude da entrada.

No sistema de regras Jess, os lugares possuem um *template* padrão, descrito abaixo:

```
(deftemplate place
   (slot name)
   (slot type (default problem))
   (multislot content))
```

Pode-se notar que além do nome e do tipo, o *template* possui um multicampo `content`. Este multicampo existe para guardar, quando da execução da RPO, as fichas que estiverem presentes no lugar.

Cada instância da classe `OpnPlace` é traduzida em JESS por um fato do tipo do *template* `place`.

### B.3.3 OpnVariable

`OpnVariable` representa uma instância global de uma classe da RPO acima definida. Serve para instanciar possíveis objetos globais, que podem ter suas propriedades usadas na definição de condições/ações das transições.

A classe `OpnVariable` possui um "ponteiro" para a classe da qual o objeto é instância, e copia desta, os campos e valores padrão, para (opcionalmente) mudar os valores.

Apesar de tudo, a declaração de objetos na definição da RPO não é usada (pelo menos atualmente, no nosso trabalho), sendo até um item opcional na gramática da rede. Por isso, a classe `OpnVariable` está marcada como `deprecated`.

### B.3.4 OpnTransition

`OpnTransition` representa uma transição da RPO. Cada transição possui um nome único, e além desta, tem como propriedades:

- Um conjunto de lugares de entrada.
- Um conjunto de classes (fichas da RPO) permitidas para cada lugar de entrada (este, usado para checagem no *parsing*).
- Um conjunto de lugares de saída.
- Um conjunto de classes para cada lugar de saída (também usado para checagem na análise sintática).
- Um conjunto (opcional) de condições para que a transição dispare.
- Um conjunto (opcional) de ações (atribuições/chamadas de métodos) que devem ser executadas logo após o disparo da transição.

Tanto as condições quanto as ações têm seus formatos especificados na gramática. Este formato permite que sejam traduzidas no formato JESS facilmente, portanto os procedimentos não estão explicitados neste documento.

Ao contrário dos lugares, que têm um *template* padrão, as transições não possuem um único *template* (observação: até poderia haver apenas um *template* padrão para todas as transições, mas isso acarretaria em uma maior complexidade na formulação das regras de disparo destas, no JESS).

Ao invés disto, gera-se dinamicamente os *templates*. Para cada grupo de transições com quantidades de lugares de entrada e saída iguais, é gerado um *template*. Por exemplo, para uma transição simples (um lugar de entrada, um lugar de saída), o *template* gerado é:

```
(deftemplate trans-1to1
   (slot name)
   (slot place-in1)
   (slot place-out1)
   (slot condition)
   (slot action))
```

Para uma transição com três lugares de entrada, e dois de saída:

```
(deftemplate trans-3to2
   (slot name)
   (slot place-in1)
   (slot place-in2)
   (slot place-in3)
   (slot place-out1)
   (slot place-out2)
   (slot condition)
   (slot action))
```

O significado da maioria dos *slots* (campos/propriedades) são auto-explicativos. Os campos `condition` e `action` contém referência para funções do JESS. No caso de `condition`, é uma função que retornará um *booleano* `TRUE/FALSE`, e é composta de um `E` lógico de todas as condições especificadas na RPO. Por exemplo, se a especificação da condição fosse:

```
Transition1: eq (batata, cebola);
...
Transition1: eq (banana, batata);
```

A função de condição da `Transition1` seria:

```
(deffunction cond_Transition1 (?token)
   (and
       (eq batata cebola)
       (eq banana batata))
)
```

E o *slot* `condition` conteria o valor `cond_Transition1`. A mesma lógica serve para as ações, com a diferença de que a função de ação não retorna valor algum, e cada ação é executada na ordem em que foi especificada na entrada. O parâmetro da função (`?token`) é um parâmetro padrão que sempre é passado para a função, e se refere a um objeto (ficha da RPO).

Estas funções de condição e de ação também são geradas dinamicamente pela classe `OpnTransition`. Se não houver condição, uma função padrão é gerada, que retorna sempre TRUE.

Cada transição tem o seu tipo de *template* gerado, e a sua representação no JESS é um fato do tipo do template.

Além disso, a classe `OpnTransition` também é responsável pela geração das regras de disparo das transições. Assim como no caso dos *templates*, as regras são específicas para cada tipo de transição ("tipo" aqui referindo-se a quantidade de lugares de entrada/saída que ela possui).

De maneira geral, a regra de disparo da transição em JESS é da forma:

¡ E ¿ = número de lugares de entrada

¡ S ¿ = número de lugares de saída

;; xxxx = linha de comentários

```
(defrule rule-trans-<E>to<S>
;;token Student
?token <- (Student (where $?token-where))

;;lugares de entrada
?in1 <- (place (name ?place-in1) (content $?conts1 & ~nil))
...
?in<E> <- (place (name ?place-in<E>) (content $?conts<E> & ~nil))

;;lugares de saída
?out1 <- (place (name ?place-out1))
...
?out<S> <- (place (name ?place-out<S>))

;;a transição
 (trans-<E>to<S>
           (place-in1 ?place-in1)
            ...
           (place-in<E> ?place-in<E>)
           (place-out1 ?place-out1)
            ...
           (place-out<S> ?place-out<S>)
```

```
            (condition ?cnd)
            (action ?act))

;;testa se o token está em todos os lugares de entrada acima
(test (member$ ?token $?conts1))
...
(test (member$ ?token $?conts<E>))

;;testa a condição especificada
(test (apply ?cnd ?token))

=>

;;cria um multicampo auxiliar com um único valor, o próprio token
(bind $?temp (create$ ?token))

;;cria lista com todos os lugares de entrada
(bind $?ins (create$ ?place-in1 ... ?place-in<E>))

;;cria lista com todos os lugares de saída
(bind $?outs (create$ ?place-out1 ... ?place-out<S>))

;;modifica o conteudo dos lugares de entrada?in<x> para ficar com a sua
;;lista de contents, menos o token q foi retirado agora
(modify ?in1 (content (complement$ $?temp $?conts1)))
...
(modify ?in<E> (content (complement$ $?temp $?conts<E>)))

;;retira do where do token, o(s) lugar(es) de entrada
(modify ?token (where (complement$ $?ins (fact-slot-value ?token where))))

;;coloca no where do token, o(s) lugar(es) de saída
(modify ?token (where (insert$ (fact-slot-value ?token where) 1 $?outs)))

;;atualiza o conteúdo do(s) lugar(es) de saida
;;se o conteudo não existir, cria, senão, insere o token no final da lista
(bind $?old_cont (fact-slot-value ?out1 content))
(if (or (eq $?old_cont nil) (eq $?old_cont (create$ nil))) then
   (modify ?out1 (content $?temp))
else
   (modify ?out1 (content (insert$ $?temp 2 $?old_cont))))
...
```

```
(bind $?old_cont (fact-slot-value ?out<S> content))
(if (or (eq $?old_cont nil) (eq $?old_cont (create$ nil))) then
   (modify ?out<S> (content $?temp))
else
   (modify ?out<S> (content (insert$ $?temp 2 $?old_cont))))

;;coloca o busy do token como FALSE, para poder disparar
(modify ?token (busy FALSE))

;;Aplica ação
(apply ?act ?token))
```

## B.4 Quarto Passo - Coordenar todos os passos anteriores

O quarto passo é implementar algo que coordene todos os passos anteriores.

Para isso, foi feita a classe `OpnTranslator`, que coordena e comanda as outras classes de estruturas de dados, e as classes geradas pelo JavaCC. Ver figura B.2.

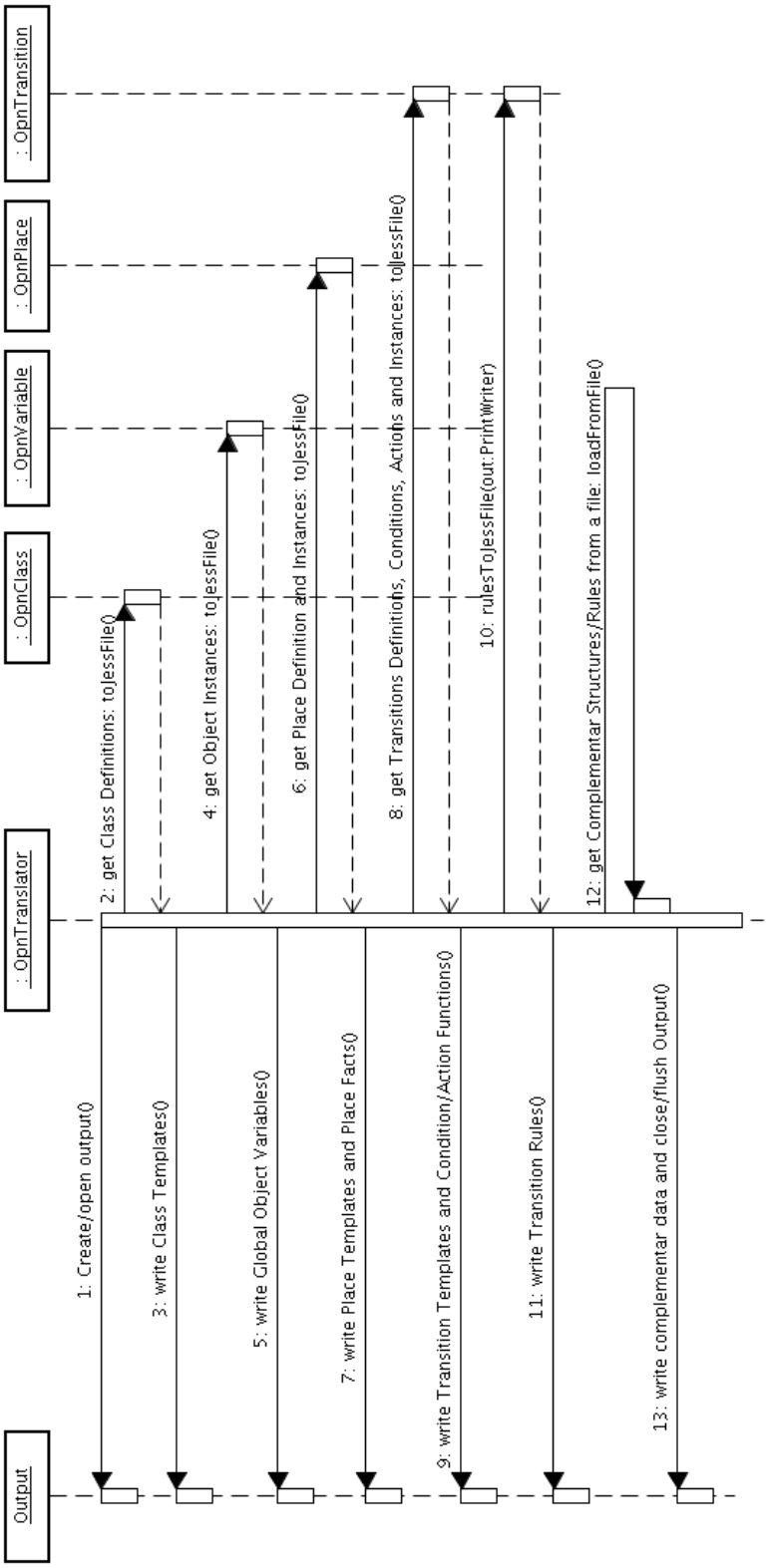



Figura B.2: Diagrama de Seqüência do processo de compilação

# Apêndice C

# Extensões da RPO - funções complementares em JESS

Como visto na seção 4.2, o modelo pedagógico do MathTutor é modelado usando Redes de Petri Objeto. Para poder "executar" a RPO, como já dito na seção 5.2, é utilizado um sistema de regras, mais especificamente o JESS. Como "ponte" entre o modelo de Redes de Petri Objeto e o sistema de regras em JESS, foi desenvolvido um compilador, cuja finalidade é transformar a RPO numa representação apropriada para ser usada no JESS (ver figura 5.2).

Entretanto, visando portabilidade de código e re-utilização deste em possíveis outros projetos, o compilador não adiciona semântica à RPO, ou seja, o sistema modelado pela RPO não tem a sua semântica automaticamente passada para o JESS. Isso quer dizer, por exemplo, que se numa determinada RPO os lugares significam diferentes tarefas a serem realizadas por uma entidade, representada por uma ficha, o sistema em JESS não consegue "dizer" para esta entidade automaticamente a tarefa a ser desempenhada. Para que isso seja possível, é necessário extender o sistema da RPO compilado em JESS, adicionando regras que reflitam as particularidades das semânticas de cada RPO.

Esta seção mostra algumas destas regras, bem como estruturas de dados que a semântica particular das RPOs de $1^o$ e $2^o$ nível do MathTutor demandaram.

## C.1 Comunicação da RPO com o ambiente externo

A RPO roda como um sistema de regras em JESS. Dentro do JESS, toda e qualquer estrutura que já tenha sido definida dentro dele é acessível, bem como classes em Java. Entretanto, somente isso não é suficiente para os nossos propósitos, pois manipular as classes Java diretamente do JESS é, além de complexo, ineficiente.

Por esses motivos, foi desenvolvido um pacote, br.ufsc.MathTutor.JessComm, que contém estruturas de dados que facilitam a comunicação com as estruturas internas do JESS. O diagrama de classes deste pacote pode ser visto na figura C.1.

O principal foco deste pacote são as classes `*JessComm`, que atualmente contam com três implementações: `AbstractJessComm`, `SyncJessComm` e `StudentJessComm`. O primeiro é uma classe abstrata, que contém métodos comuns a todas classes que desejam se comunicar com o JESS.

Entretanto, o código dentro do JESS também deve ser preparado para a comunicação com o JessComm. Para isso, a seguinte linha deve constar do código em JESS:

```
(defglobal ?*jessComm* = null)
```

O código acima declara uma variável global de nome `?*jessComm*`. Essa variável conterá uma referência para um objeto que extenda `AbstractJessComm`. Entretanto, é necessário enviar ao JESS a referência deste objeto. Para isso, na classe `AbstractJessComm` existe o método `setJessComm()`, que passa para dentro do JESS a instância do objeto. Este método deve ser chamado antes de se iniciar a execução das regras.

A classe `SyncJessComm` herda de `AbstractJessComm`, e implementa uma comunicação síncrona do ponto de vista das regras do JESS, isto é, a execução do JESS fica bloqueada até uma resposta chegar. Do ponto de vista de um programador JESS, é uma chamada de função como um “read”. Essa abordagem foi a primeira adotada para o projeto, mas foi descartada por não oferecer suporte a múltiplas fichas concorrentes na RPO. A abordagem pode ser vista no diagrama de seqüência na figura C.2.

De dentro do JESS, o codigo seria:

```
(call ?*jessComm* ask <parametros>)
```

E a execução do JESS ficaria parada até o retorno de um valor.

A classe `StudentJessComm` também herda de `AbstractJessComm`, mas implementa uma comunicação assíncrona. Este mecanismo é mais complicado que o anteriormente descrito, e envolve uma outra classe, a `OpnStudentJessCommBean`.

Na comunicação assíncrona, assim como na síncrona, no código em JESS, faz-se uma chamada a uma função do `JessComm`, igual ao código síncrono acima. A diferença é que esta função é não-bloqueante, ou seja, ela não retorna valor algum, e a execução do JESS continua normalmente. A verdadeira resposta a esta função é retornada por outro meio (e possivelmente, depois de algum tempo).

Por questão de implementação, os métodos da máquina de inferência do JESS são sincronizados, ou seja, bloqueantes. Isso significa que, ao executarmos um `run` (execução das regras), não conseguimos executar outros comandos, como `assert` (para inserir fatos na base de dados). Portanto, não podemos usar nenhuma das funções do JESS para inserir uma resposta assíncrona, de uma chamada de função.

Necessita-se de alguma regra que seja responsável por “perceber” a existência de resposta, e então recebê-la e entregá-la ao devido lugar. Entretanto, como receber um evento externo (percepção) se

as funções JESS não são executadas até a execução das regras terminar? Felizmente, o JESS possui suporte a arquitetura JavaBeans.

O JESS possui a capacidade de perceber a mudança de propriedades de um JavaBean, se este tiver suporte para sinalizar mudanças de propriedades por eventos. Utilizando-se desse artifício, escrevemos uma regra em JESS, tendo como idéia: ao se mudar uma propriedade no JavaBean, sinaliza que há uma resposta, e então o JESS pode "perguntar" ao JavaBean, a resposta.

Para o caso do nosso trabalho, foi desenvolvido um JavaBean específico, a já citada classe `OpnStudentJessCommBean`. Este contém muito da informação e semântica da ficha da nossa RPO, da classe `Student`.

Assim como no caso do `JessComm`, para funcionar o esquema acima descrito, o código JESS deve ter algumas linhas que o preparem:

```
;;importa lib
(import br.ufsc.MathTutor.JessComm.*)

;;definição de template da classe do java bean usado para
;;comunicação assincrona
(defclass JessCommBean OpnStudentJessCommBean)

;;variavel global - bean de comunicação assincrona
(defglobal ?*jessCommBean* = null)
```

E igualmente ao `JessComm`, há a necessidade de se "passar" para o JESS, a instância do objeto JavaBean. A classe `StudentJessComm`, no seu método `setJessComm()`, já está configurada para isso.

Dentro do código JESS, a regra que é responsável por esta comunicação é algo como:

```
(defrule answer
?bean <- (JessCommBean (flag 1) (ID ?id))
?token <- (Student (ID ?id) (doing_IU ?doing))
=>
;;pega o objeto
(bind ?obj (fact-slot-value ?bean OBJECT))

;;pega a resposta
(bind ?asw (call ?obj getValueAnswer))

;;processa a resposta de alguma maneira .......
```

```
;;muda a flag, sinalizando que esta resposta já foi "consumida"
(?obj setFlag 0)
)
```

Na primeira linha, temos um "*match*" do objeto JavaBean, com o *template* JessCommBean (definido com o `defclass` no código anterior). Note o campo `flag`, tendo como valor 1. Esse valor foi arbitrariamente decidido, e neste caso, significa existência de resposta. E para identificar qual foi a ficha (fato do tipo *template* `Student`) que anteriormente havia chamado uma função requisitando resposta, há o campo `ID`, que identifica uma ficha unicamente nesta RPO.

Por uma questão de *type casting* (verificação de tipos dinamicamente) intrínseco do JESS, não recuperamos o valor da resposta diretamente do *template* gerado pelo JavaBean, mas sim do próprio objeto JavaBean, utilizando uma chamada da função `call (call ?obj getValueAnswer)`.

Sendo uma RPO dinâmica onde as fichas não são todas criadas na marcação inicial, necessita-se um meio de inserir as fichas na rede também dinamicamente. Para isso, também usamos o `OpnStudentJessCommBean`, com a seguinte regra (simplificada):

```
(defrule newToken
?bean <- (JessCommBean (flag 2) (name ?name) (ID ?id))
=>
;;pega o objeto bean
(bind ?obj (fact-slot-value ?bean OBJECT))


;;chama os metodos que devolvem valores para preencher o token Student
(bind ?doing_IU (call ?obj getValueDoing_IU))
(bind ?doing_Pb (call ?obj getValueDoing_Pb))
;;......

;;insere o token Student na base de dados
(bind ?stu (assert (Student (ID ?id)
                            (name ?name)
                            (doing_IU ?doing_IU)
                            (doing_Pb ?doing_Pb)
                            ;;todos os campos...
)))

;;sinaliza objeto bean que os dados já foram consumidos
(?obj setFlag 0)

;;verifica onde o token deve estar (slot where) e atualiza
;;os conteúdos (content) dos lugares
```

```
;;......

;;seta outras estruturas, se necessário
;;....
)
```

O processo de comunicação assíncrona se encontra descrito no diagrama de seqüência da figura C.3.

## C.2 Implementação da semântica da RPO de primeiro nível

Na RPO de 1º nível, cada lugar representa um Problema de uma Unidade Pedagógica a ser feito pelo Estudante. Entretanto, este problema desmembra-se nas Unidades de Interação (explicação, exemplo e exercício), que é de domínio da RPO de 2º nível. Portanto, a semântica de um lugar na RPO de 1º nível é apenas transferir o Estudante para as Unidades de Interação da RPO de 2º nível, e esperar o seu término ali.

Para isso, foi implementada uma regra em JESS, que segue abaixo:

```
(defrule place
?in <- (place (name ?place-in) (content $?conts & ~nil)
              (type problem))
?token <- (Student (done $?done) (where $?where)
                   (doing_Pb ?place-in) (busy FALSE))

;;testa pra ver se o lugar pertence ao where do Student
(test (neq (member$ ?place-in $?where) FALSE))

;;testa para ver se o lugar não pertence ao done do Student
(test (eq (member$ ?place-in $?done) FALSE))
=>

;;indica que não eh para disparar com esse token de novo
(modify ?token (busy TRUE))

;;chama função interface
(interface ?token)
)
```

Pode-se ver que a regra não é complexa, apenas modificando um campo da ficha (*token* - para que este não dispare mais de uma vez para um mesmo lugar), e chamando a função `interface`, passando a ficha como parâmetro.

A função `interface` é abaixo transcrita:

```
(deffunction interface (?stu)
   (bind ?name (fact-slot-value ?stu name))
   (bind ?id (fact-slot-value ?stu ID))
   (bind ?curriculum (fact-slot-value ?stu doing_Curriculum))
   (bind ?pb (fact-slot-value ?stu doing_Pb))

   (call ?*jessComm* ask ?id ?name ?curriculum ?pb)
)
```

Pode-se ver que a função interface também é muito simples, sendo em essência, apenas a chamada de um método do `JessComm`. No trabalho, a classe `StudentJessComm` implementa um método `ask`, com os parâmetros específicos que a função acima pede. Este método no `StudentJessComm` chama outro método, de uma entidade externa (interface `ExternalEntity` - implementada pelo agente), que esteja se comunicando com o JESS.

Além da regra de lugar na RPO, outra questão na RPO de 1º nível a ser abordada é a questão de próximo problema a ser executado. Em uma transição, uma ficha pode ser enviada para mais de um lugar (e esses lugares são teoricamente concorrentes). Entretanto, essa concorrência neste trabalho não existe, pois na verdade, o estudante sempre estará executando apenas um problema por vez (indicado pelo campo `doing_Pb`). Para tanto, há de se decidir uma ordem para quando houver mais de um problema possível de ser feito.

Na definição da RPO , algumas transições têm como ação, modificar o campo `doing_Pb` da ficha chamando uma função, `Next_Problem`.

Esta função atualmente está implementada de forma simples, recebendo como parâmetro apenas a própria ficha, e retornando um problema qualquer que ainda não tenha sido feito. Futuramente, mecanismos mais complexos para esta função, como levar em conta preferências, etc., serão implementados.

Outras questões da RPO de 1º nível, aqui não abordadas, são comuns à RPO de 2º nível. Essas questões serão abordadas junto com as peculiaridades desta última rede.

## C.3 Implementação da semântica da RPO de segundo nível

Na RPO de 2º nível, os lugares ou apresentam uma Unidade de Interação (UI), ou um lugar de semântica de controle. Os lugares de UI são: Explanation, Example e Exercise, e os de controle são Begin, Halt e End. A regra para os lugares de UI é:

```
(defrule place
```

```
?in <- (place (name ?place-in & ~Halt & ~End)
              (content $?conts & ~nil))
?token <- (Student (answer nil) (busy FALSE))

;;testa pra ver se o lugar pertence ao where do Student
(test (neq (member$ ?place-in (fact-slot-value ?token where)) FALSE))

=>
;;indica que não é para disparar com esse token de novo
(modify ?token (busy TRUE))

;;chama função interface
(interface ?token)
)
```

Esta regra é bem simples, vemos que a ficha ainda não tem nenhuma resposta (`answer nil`). Este campo guarda a resposta da interface, que pode ser tanto uma nota de um exercício, como a preferência por qual unidade de interação prosseguir, além de mensagem de saída do sistema (`halt`). Observa-se ainda que a regra é a mesma para todos os lugares exceto os lugares de controle Halt e End (a explicação da regra destes lugares está mais adiante).

Podemos ver que esta regra tem o lado direito (RHS) igual a regra de lugar da RPO de 1º nível. A função interface também é similar. O grande diferencial está, na verdade, na Entidade Externa (`ExternalEntity` - o agente), que trata cada rede de modo peculiar.

Na RPO de 2º nível, são usadas algumas estruturas de dados que não são usadas na RPO de 1º nível (apesar de existirem ali também, ainda não são usadas). Estas estruturas referem-se a Relatórios (*Reports*), sobre as atividades de cada Estudante no decorrer de suas interações:

```
(deftemplate PB_Report
  (slot name_Pb)
  (slot student_id)
  (multislot rep_Exp)
  (multislot rep_Exa)
  (multislot rep_Exe)
)

 (deftemplate Exe_Report    ;;Relatório de Exercício
  (slot name_Pb)
  (slot student_id)
  (slot count_views (default 0))
  (slot grade (default 0.0))
  (slot name)
```

```
)

(deftemplate Exa_Report     ;;Relatório de Exemplo
  (slot name_Pb)
  (slot student_id)
  (slot count_views (default 0))
  (slot name)
)

(deftemplate Exp_Report     ;;Relatório de Explicação
  (slot name_Pb)
  (slot student_id)
  (slot count_views (default 0))
  (slot name)
)
```

Os templates `ExX_Report` representam um relatório de um Estudante em uma unidade de interação. Eles são praticamente iguais, exceto pelo `Exe_Report` (relatório de Exercício), que tem um campo indicando a nota conseguida. O *template* `PB_Report` agrega todos os relatórios de um Estudante para um determinado problema. A ficha de *template* `Student` contém um multicampo `report`, onde armazena uma lista com os `PB_Reports`.

Estas estruturas são criadas e/ou atualizadas quando há o disparo de transições, assim especificado nas ações destas (na chamada da função `Add_Report`).

Além destas, a RPO de 2º nível exige uma estrutura de dados exclusiva, mostrada abaixo:

```
(deftemplate PB_Contents
  (slot student)
  (slot problem)
  (multislot exp)
  (multislot exp_done)
  (multislot exa)
  (multislot exa_done)
  (multislot exe)
  (multislot exe_done)
)
```

Esta estrutura se faz necessária por causa da maneira que a RPO de 2º nível trabalha: existem apenas três lugares de Unidade de Interação (Explicação, Exemplo e Exercício). Entretanto, para cada uma delas, há um número não fixo de unidades, ou seja, podemos ter várias explicações diferentes de um mesmo assunto, por exemplo.

PB_Contents possui multicampos onde guarda listas de explicações, exemplos e exercícios, de um determinado problema para um estudante. Além disso, mantém uma lista de explicações, exemplos e exercícios já executados.

A RPO de 2º nível no entanto, no seu atual estado, permite que um estudante faça vãrias vezes uma mesma Unidade de Interação. Isso constitui um problema, e para resolvê-lo, foram tomadas duas decisões: se não houver mais uma UI para ser feita (ou seja, se o estudante escolheu ver mais uma explicação, e não houver mais), a RPO retorna null, indicando o fato de não haver mais disponíveis. Se o estudante quiser no entanto, pode repetir os conteúdos; nesse caso, PB_Contents é configurado (para determinada UI) como se o estudante estivesse começando.

PB_Contents é usado pelas funções de próxima unidade de interação (Next_IX), quando do disparo de uma transição que leva a determinado lugar. Por exemplo, quando uma transição leva ao lugar Exercise da RPO, uma das ações desta transição é atribuir ao campo doing_IU da ficha Student, o valor da função Next_IE. Esta função é mostrada de forma simplificada abaixo:

```
(deffunction Next_IE (?token)
   ;;chama função que recupera PB_Contents deste token,
   ;;e atribui a variável ?tobedone
   (bind ?tobedone (Recover_PB_Contents ?token))

   ;;atribui a variável ?next o valor da primeira entrada
   ;;do multicampo exe de PB_Contents
   (bind ?next (nth$ 1 (fact-slot-value ?tobedone exe)))

   ;;se ?next for nulo...
   (if (eq ?next nil) then
       ;;insere em exe, todo o conteúdo de exe_done
       ;;e deixa exe_done vazio
       ;;...
   ;;se ?next não for nulo...
   else
      ;;insere ?next em exe_done
      ;;e retira ?next de exe
      ;;...
   ))

   ;;retorna ?next
   (return ?next)
)
```

Atualmente, PB_Contents tem seus dados inicialmente atribuídos de maneira estática, ou seja, no próprio código JESS (quando da inserção de uma nova ficha - ver regra newToken desta rede). En-

tretanto, no sistema real (não implementado ainda) a prática será ler quais conteúdos estão presentes para o problema, e preencher os campos de `PB_Contents` de maneira a refletir isso.

A última regra a mencionar na RPO de 2º nível é a regra para os lugares Halt e End. Estes lugares representam a vontade expressa pela Entidade Externa de parar, ou então a conclusão do problema (atualmente, pelo recebimento de uma nota maior que 7 em um dos exercícios). Ao contrário dos outros lugares desta rede, nestes dois, não há uma mensagem para a Entidade Externa pedindo por uma resposta, mas sim uma mensagem informando o término da rede para determinado Estudante.

A regra (simplificada) segue abaixo:

```
(defrule placeHaltEnd
?in <- (place (name Halt|End) (content $?conts & ~nil))
?token <- (Student (ID ?id) (doing_Pb ?pb))

;;testa pra ver se o lugar pertence ao where do Student
(test (neq (member$ (fact-slot-value ?in name)
                    (fact-slot-value ?token where)) FALSE))

=>

;;chama função de "salvamento"
(saveStudent ?token)

;;remove o ?token da rede e todos os objetos relacionados,
;;como PB_Content, e reports

;;e por fim, retira o proprio student da base
(retract ?token)

;;manda msg para jessComm
(bind ?place (fact-slot-value ?in name))
(call ?*jessComm* send (create$ ?id ?place ?pb))
)
```

Observa-se que a regra trata de salvar o estado da ficha `Student` (na função `saveStudent ?token`), e depois retira da base de conhecimento todos os fatos relacionados ao Estudante. Por fim, envia uma mensagem à Entidade Externa, informando o término da rede com o determinado Estudante.

A função `saveStudent` na verdade, apenas chama os métodos do `StudentJessComm`, enviando para este como parâmetros, os campos dos *templates*. Basicamente, o `StudentJessComm` possui dois métodos de persistência: `save` e `saveReport`. O primeiro salva as informações do *template* `Student`, e o segundo, salva as informações dos relatórios.

A função `saveStudent` é comum a RPO de 1º nível também.

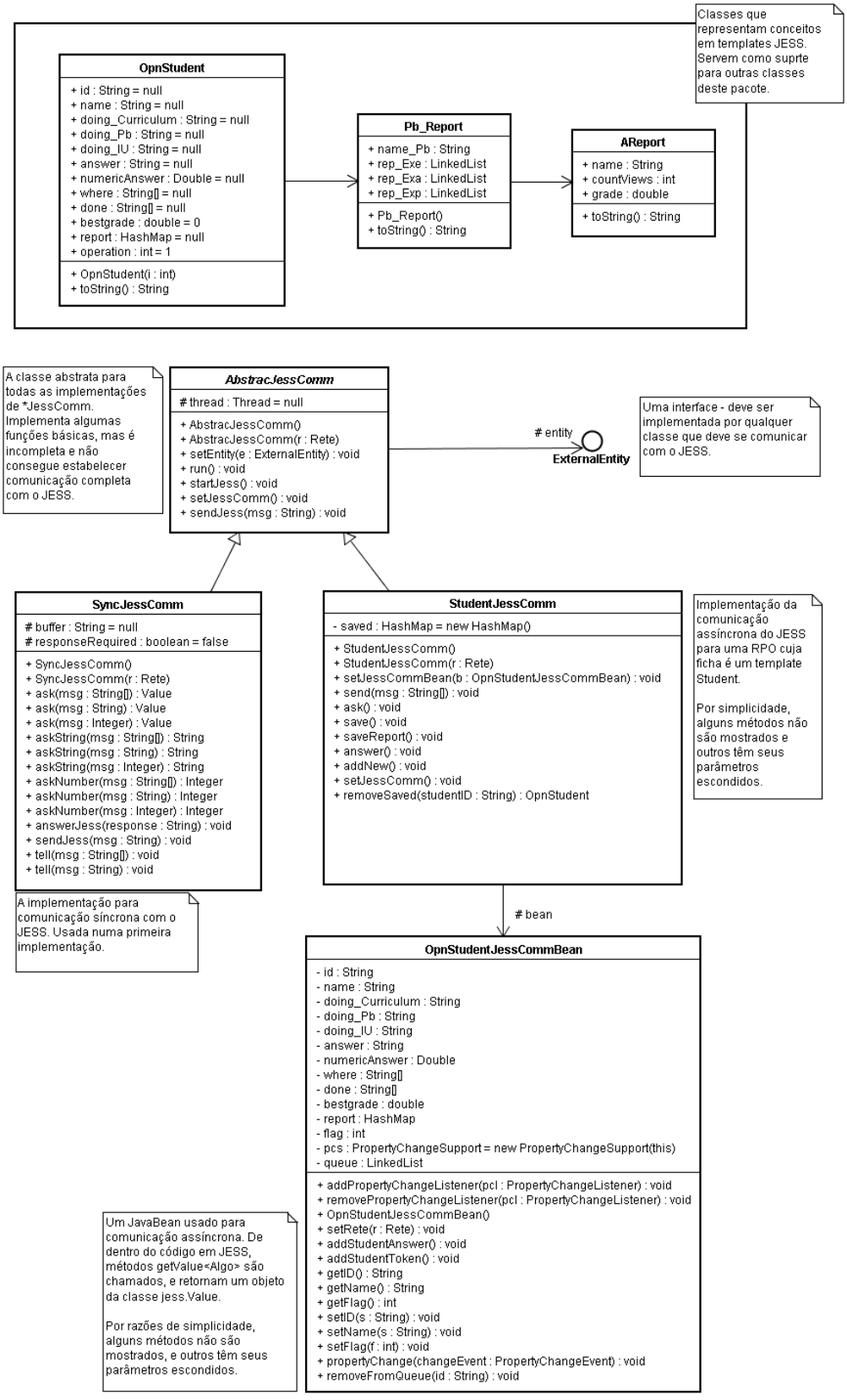



Figura C.1: Diagrama de Classes do pacote br.ufsc.MathTutor.JessComm

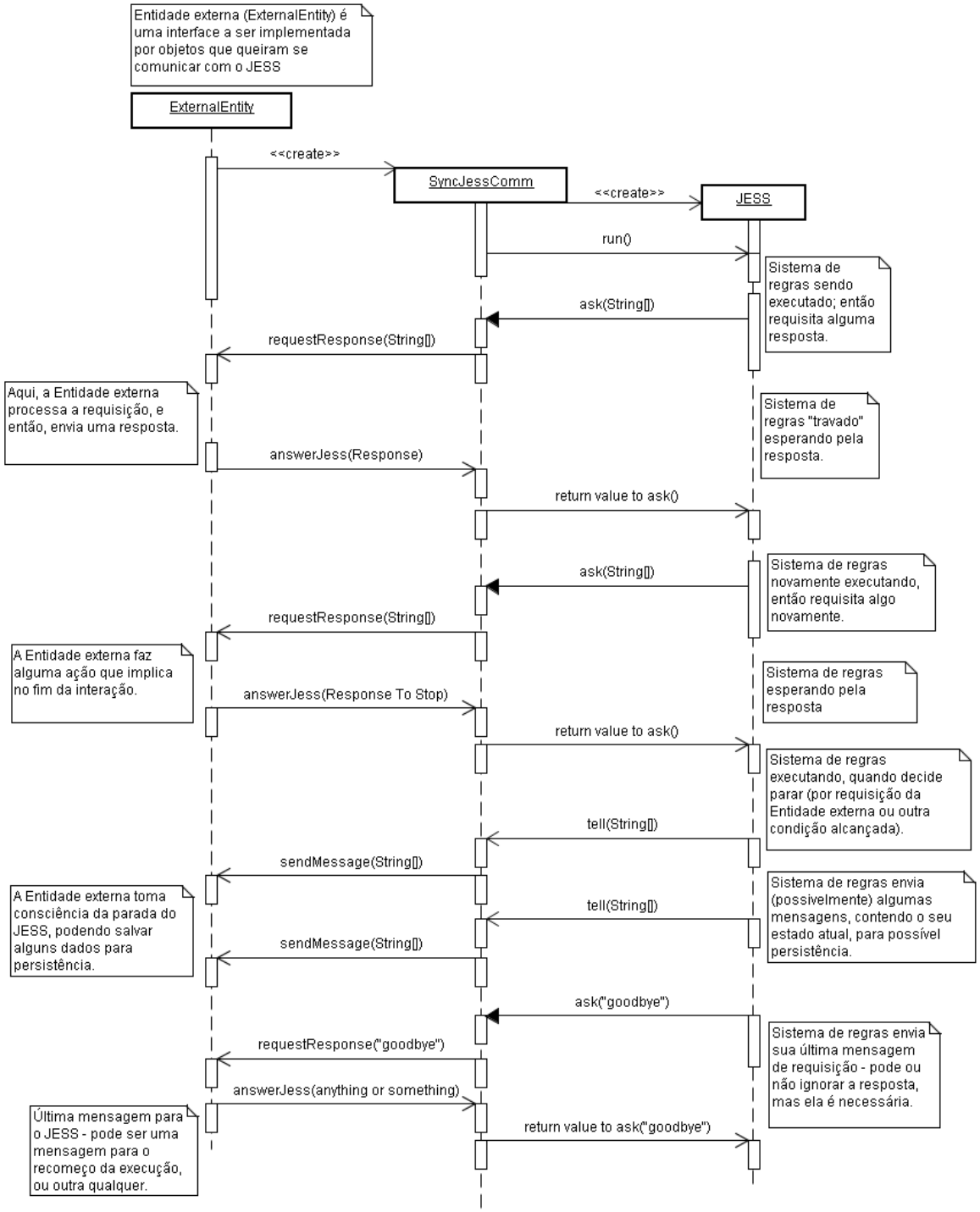
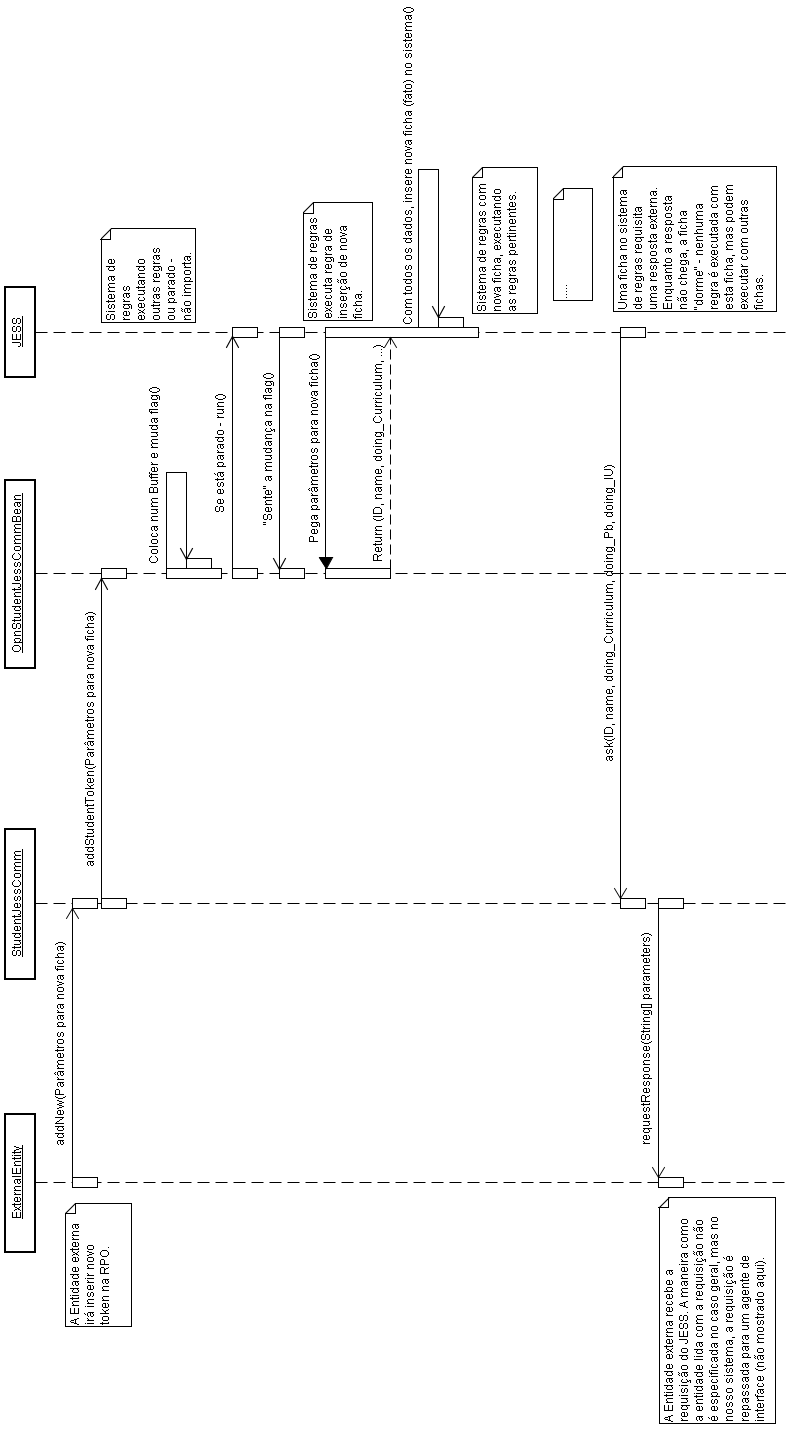



Figura C.2: Diagrama de Seqüência de comunicação síncrona entre JESS e algum objeto externo


JESS
OpnStudentLessCommBean
StudentLessComm
ExternalEntity
Sistema de regras executando outras regras ou parado - não importa.
Sistema de regras executa regra de inserção de nova ficha.
Com todos os dados, insere nova ficha (fato) no sistema()
Sistema de regras com nova ficha, executando as regras pertinentes.
.....
Uma ficha no sistema de regras requisita uma resposta externa. Enquanto a resposta não chega, a ficha "dorme" - nenhuma regra é executada com esta ficha, mas podem executar com outras fichas.
Coloca num Buffer e muda flag()
Se está parado - run()
"Sente" a mudança na flag()
Pega parâmetros para nova ficha()
Return (ID, name, doing_Curriculum, ...)
addStudentToken(Parâmetros para nova ficha)
ask(ID, name, doing_Curriculum, doing_Pb, doing_IU)
addNew(Parâmetros para nova ficha)
requestResponse(String[] parameters)
A Entidade externa irá inserir novo token na RPO.
A Entidade externa recebe a requisição do JESS. A maneira como a entidade lida com a requisição não é especificada no caso geral, mas no nosso sistema, a requisição é repassada para um agente de interface (não mostrado aqui).

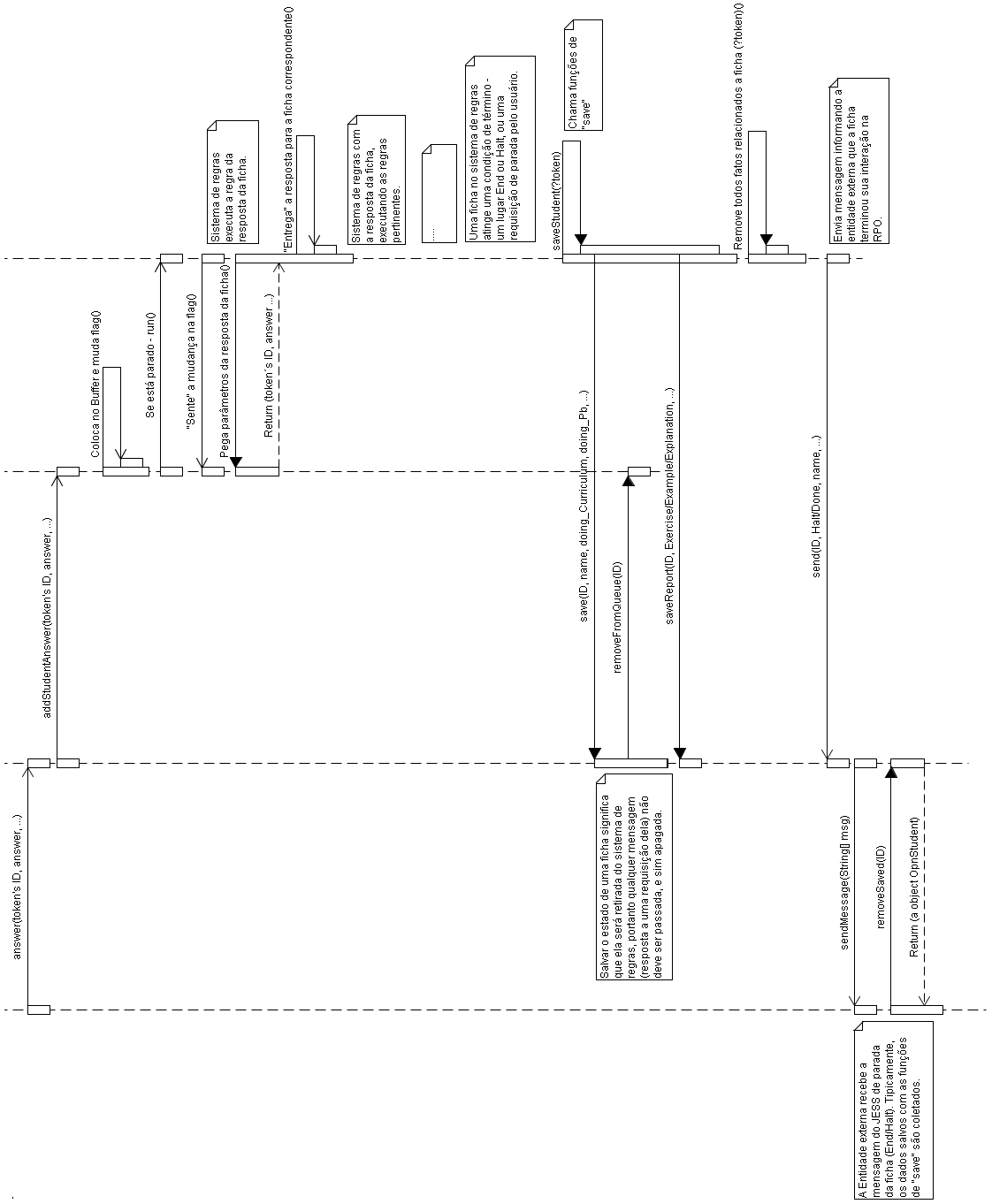



Figura C.3: Diagrama de Sequência da Comunicação assíncrona entre JESS e objeto externo - parte2

# Apêndice D

# Implementação de um agente em JADE para gerenciar as RPOs de controle do modelo pedagógico

Como visto em 4.1, um dos módulos da arquitetura do MathTutor é a Sociedade de Agentes Tutores Artificiais (SATA - ver figura 4.1).

Como já citado em 5.1, foi implementado um agente tutor. O sistema tutor contido neste agente é controlado por duas Redes de Petri Objeto, conforme visto em 4.2. Uma das principais funções do agente tutor é controlar e coordenar os dois níveis de RPOs contidos em si.

Nesta seção são apresentados os detalhes de implementação deste agente na plataforma JADE, que contém dentro de si dois *engines* JESS, um para cada nível das RPOs.

Um agente em JADE, se não tiver uma interface gráfica (como neste caso), deve herdar da classe `jade.core.Agent`. Os dois principais métodos que devem ser implementados são: `setup()` e `takeDown()`. O primeiro é usado para inicialização do agente, e é onde são instanciadas as estruturas de dados usadas, além de adição de *behaviours* (comportamentos), e configurações com o ambiente. O método `takeDown()` é invocado quando o agente está para ser desativado (descartado), sendo usado para "limpeza" do agente (retirada de registro no ambiente, etc).

Os `Behaviour`s acima mencionados são os meios pelo qual o agente executa uma tarefa. São chamados de comportamentos, e implementam ações do agente. É por meio dos `Behaviour`s que os agentes em JADE executam algo (implementando o método `action()` dos `Behaviour`s.).

O agente que mantém as RPO deste trabalho é o `StudentPetriNetsAgent`. Abaixo, a sua assinatura:

```
public class StudentPetriNetsAgent extends jade.core.Agent
```

Internamente, ele possui as seguintes estruturas:

```
private StudentJessComm jessComm1;
private StudentJessComm jessComm2;
private HashMap studentsID1 = new HashMap();
private HashMap studentsID2 = new HashMap();
```

Os dois primeiros são objetos que tratam da comunicação com o JESS. O primeiro JESS rodará um sistema de regras equivalente a RPO de 1.o nível, e o segundo JESS, a RPO de 2.o nível.

Note que há dois `HashMap`s, de nome `studentsID#`. Cada `HashMap` contém um par de `String`s, sendo que a chave (`key`) é o ID do Estudante. O valor (`value`) é: o problema (`doing_Pb`) no caso da RPO de 2.o nível, ou o currículo (`doing_Curriculum`) no caso da RPO de 1.o nível. Essas estruturas se fazem necessárias para que mensagens oriundas de agentes externos, caso cheguem com atraso (ou com informação incorreta, por algum motivo) não sejam entregues às RPOs. Atraso no sentido de que a ficha pode ter terminado a execução na rede, e uma resposta anterior para ela ainda não ter sido entregue.

Caso isso acontecesse, o funcionamento da RPO seria afetado, e na atual implementação, causaria uma espera indefinida (*deadlock*). Isso porque o `OpnStudentJessCommBean`, que atua como um *buffer* de mensagens, é implementado como uma fila. E no caso de uma mensagem que não tenha um destinatário na RPO, essa mensagem ficaria parada e a fila bloquearia, causando uma espera. Como nenhuma instância de ficha da rede nunca receberia essa mensagem, eventualmente todas as fichas travariam requisitando respostas, que ficariam para sempre na fila.

Para os `Behaviour`s, que são os objetos que tratam diretamente com os `StudentJessComm`, estarem cientes se uma mensagem pode ou não ser entregue, o `StudentPetriNetsAgent` implementa os seguintes métodos para interagir com os `HashMap`s acima apresentados:

```
public void addStudentID(int net, String id, String doing);
public boolean removeStudentID(int net, String id, String doing);
public boolean hasStudentID(int net, String id, String doing);
```

O agente `StudentPetriNetsAgent` tem mecanismos (detalhados adiante) para gerenciar mensagens entre as duas RPO. Entretanto, existem ocasiões onde a RPO exige uma resposta externa (tipicamente, da interface com o Estudante). Para isso, o agente deve se comunicar com outro(s) agente(s), a fim de obter a resposta. Atualmente, é necessário apenas um tipo de agente, o agente de interface. A referência para ele é dada em um objeto do tipo `jade.core.AID` (*Agent ID*). Como o agente pode se comunicar com vários agentes de interface ao mesmo tempo, estas referências ficam armazenadas no `HashMap`:

```
private HashMap interfaceAgents = new HashMap();
```

Este `HashMap` usa como chave uma `String` representando a identificação (`ID`) do estudante, e guarda como valor, a identificação do agente (`AID`).

Os Behaviours que precisarem interagir com os agentes de interface, podem conseguir referências para estes, com os seguintes métodos da classe StudentPetriNetsAgent, passando como argumento, a identificação do estudante:

```
public AID getInterfaceAgent(String id);
public void setInterfaceAgent(String id, AID aid);
```

A classe StudentPetriNetsAgent possui três principais Behaviours implementados, e que são inseridos no método setup():

```
class OpnFirstLevelMessageBehaviour extends
                                OpnInternalMessageBehaviour
class OpnSecondLevelMessageBehaviour extends
                                OpnInternalMessageBehaviour
class OpnExternalMessageBehaviour extends CyclicBehaviour
```

Sendo que OpnInternalMessageBehaviour é uma classe abstrata, que serve de apoio para as duas primeiras classes acima. Na figura D.1, um diagrama de classes das referidas classes.

Apesar do agente StudentPetriNetsAgent ter como atributos (ou propriedades) dois objetos StudentJessComm, ele não implementa a interface br.ufsc.MathTutor.JessComm.ExternalEntity, interface esta que indica a possibilidade de comunicação de dentro do JESS para/com uma classe em Java. Isso quer dizer que quando as RPOs requisitarem algo de uma Entidade Externa (externa para elas), não será para o agente diretamente. Escolheu-se implementar a interface ExternalEntity num Behaviour, pois são os Behaviours que executam ações nos agentes:

```
abstract class OpnInternalMessageBehaviour extends CyclicBehaviour
                                        implements ExternalEntity
```

A interface ExternalEntity exige a implementação de dois métodos:

```
public void requestResponse(String[] msg);
public void sendMessage(String[] msg);
```

Sendo o primeiro usado pelo JessComm para requisitar uma resposta a uma mensagem, e a segunda, usada para enviar uma mensagem que não se espera resposta.

Na implementação de OpnInternalMessageBehaviour, esses dois métodos apenas adicionam uma posição no vetor (primeira posição) que contém a mensagem, indicando o tipo de mensagem

(“request” ou “message”), e adicionam este vetor numa fila. O tratamento destas mensagens se dá nas implementações concretas de OpnInternalMessageBehaviour, que examinaremos abaixo.

Comecemos com o Behaviour OpnFirstLevelMessageBehaviour. Este, como o nome indica, trata das mensagens vindas da RPO de 1.o nível. O método action() deste Behaviour retira uma mensagem da fila, e verifica se é uma requisição de resposta (request) ou uma mensagem simples (message). Se for o caso desta última, é indicação de que uma ficha Student terminou a execução na RPO. Na atual implementação, como os dados ainda não estão sendo enviados para algum meio persistente, simplesmente retira-se da RPO todas as referências para o Estudante.

No caso da mensagem vinda da RPO de 1.o nível ser uma requisição (request), indica que a ficha deve ser enviada para a RPO de 2.o nível para interação. A resposta a essa requisição será recebida depois do Estudante terminar a sua interação com a RPO de 2.o nível. A inserção da ficha na RPO de 2.o nível é executada diretamente no action() deste Behaviour, pois apesar de receber apenas requisições da RPO de 1.o nível, ele possui referências para as duas redes.

O Behaviour OpnSecondLevelMessageBehaviour, como o nome sugere, trata das mensagens oriundas da RPO de 2.o nível.

Se a mensagem for uma mensagem simples, ou seja, não requisitar resposta, a RPO de 2.o nível está informando que um Estudante, ou terminou a interação na rede, atingindo a condição estabelecida para que possa sair de determinado problema, ou então o Estudante, através da interface, escolheu parar a interação com o sistema (resposta halt). Em ambos os casos, o procedimento é o mesmo: retira-se todas as referências do Estudante da rede, e envia para a RPO de 1.o nível a resposta do request que originou a ficha na RPO de 2.o nível, informando claro, se a interação terminou por um *logout* do Estudante (Halt) ou por fim da rede (End).

Entretanto, se a RPO requisitou uma resposta (request) para o OpnSecondLevelMessageBehaviour, então este deve pedir para um agente externo, que forneça o serviço de interface, uma resposta para o request da RPO. Para isso, é utilizado outro Behaviour, *inner class* de OpnSecondLevelMessageBehaviour, denominado InterfaceInitiator:

```
private class InterfaceInitiator extends SimpleBehaviour
```

Este Behaviour, que extende a classe SimpleBehaviour, segue o protocolo FIPA-Request para requisitar a resposta ao agente de interface. Apesar do JADE oferecer algumas classes de Behaviours “pré-prontas” para o uso desse protocolo, alguns comportamentos não bem documentados destas classes motivaram a implementação particular do protocolo neste Behaviour.

O resultado deste Behaviour (se não acontecer erro na comunicação) é a resposta dada pelo usuário, que deve ser entregue à RPO de 2.o nível. Essa resposta tem sua chegada via uma mensagem, tratada por este mesmo Behaviour, uma vez que faz parte do protocolo, sendo a última mensagem (de tipo - *performative* no jargão FIPA - INFORM).

Outro Behaviour que recebe mensagens externas de outros agentes, é OpnExternalMessageBehaviour, já citado acima. Este Behaviour aceita mensagens de tipo

(*performative*) `REQUEST`, ou seja, aceita requisições simples ou de início de protocolos, via uma "máscara", ou *template*:

```
ACLMessage msg = myAgent.receive
        (MessageTemplate.MatchPerformative(ACLMessage.REQUEST));
```

Por enquanto protocolos mais avançados ainda não foram implementados para o sistema multiagente. Por isso, a única requisição implementada neste `Behaviour` é a inserção de uma ficha na RPO de 1.o nível, ou seja, a inserção de um estudante executando o currículo, seja o estudante continuando interação passada, ou um estudante novo.

Se o campo language indicar "`application/javaobject`", isso quer dizer que um agente externo está requisitando a este agente que insira na RPO de 1.o nível, uma ficha. Os valores desta ficha (de tipo `Student`) é passado para o agente na forma de um objeto serializado da classe `br.ufsc.MathTutor.JessComm.OpnStudent`. Essa forma, a troca de objetos serializados através de mensagens `ACLMessage`, não é encorajado pelos desenvolvedores do JADE, uma vez que esta técnica não é padronizada pela FIPA. Futuramente, com a implementação das ontologias desenvolvidas e dos protocolos, esta parte deverá ser reescrita/adaptada.

A figura D.2 contém um diagrama de seqüência contendo as interações internas do agente `StudentPetriNetsAgent`, e a figura D.3 contém um diagrama de estados descrevendo de forma simplificada, os estados do agente.

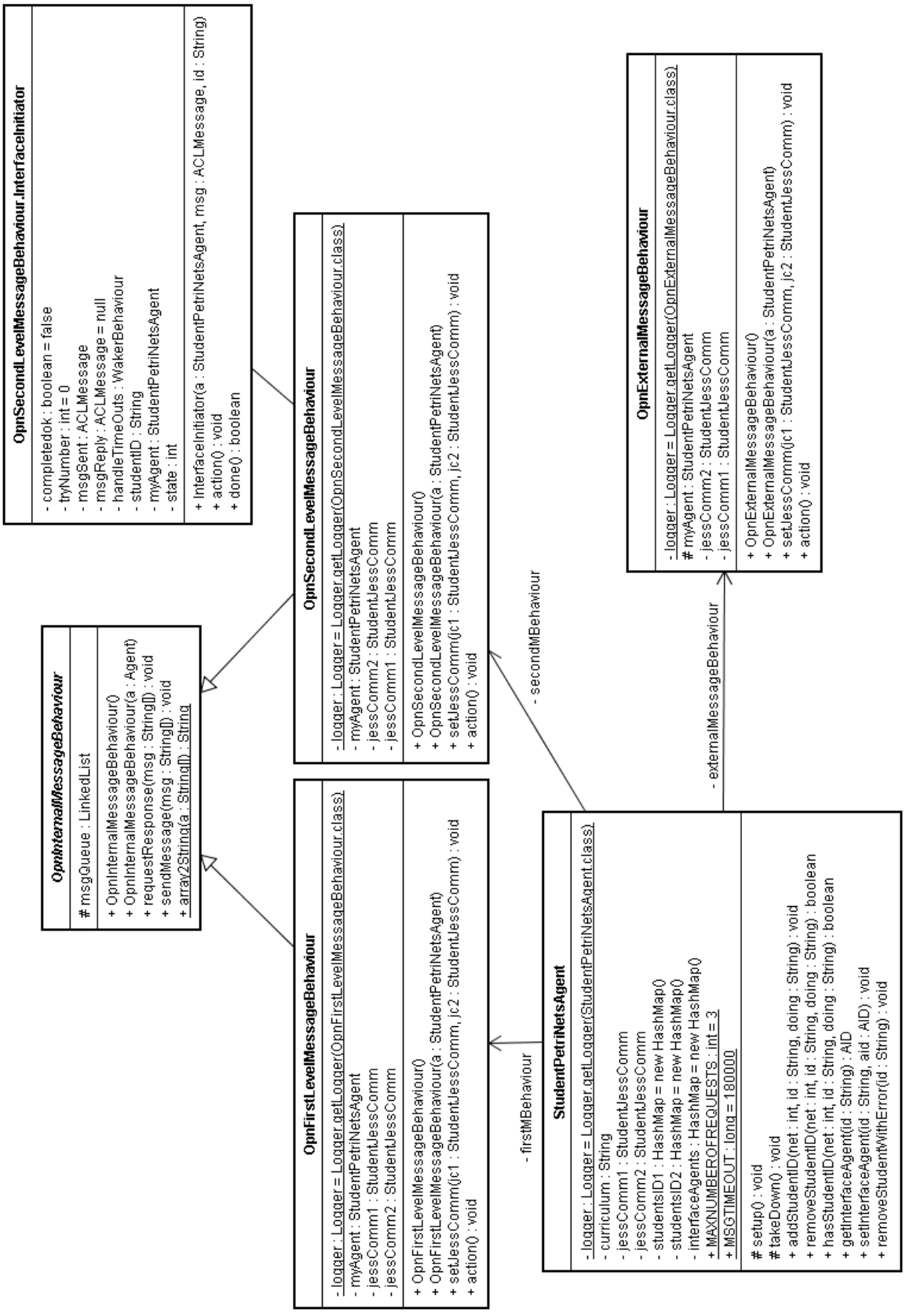
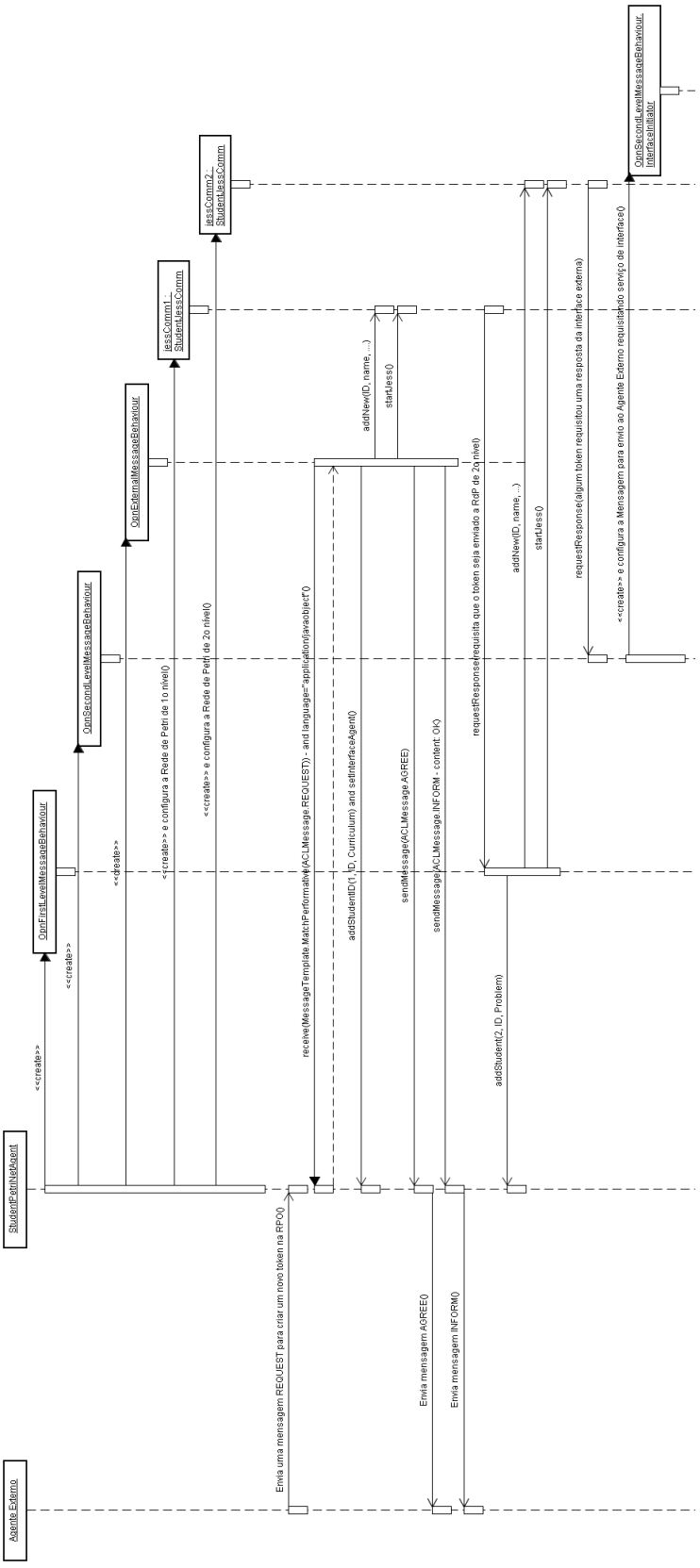



Figura D.1: Diagrama de Classes do pacote `br.ufsc.MathTutor.PetriNetsAgent`


StudentPetriNetAgent
Agente Externo
OpnFirstLevelMessageBehaviour
OpnSecondLevelMessageBehaviour
OpnExternalMessageBehaviour
lessComm1 : StudentLessComm
lessComm2 : StudentLessComm
OpnSecondLevelMessageBehaviour : InterfaceInitiator
<<create>>
<<create>>
<<create>> e configura a Rede de Petri de 1o nível()
<<create>> e configura a Rede de Petri de 2o nível()
receive(MessageTemplate.MatchPerformative(ACLMessage.REQUEST)) - and language="application/javaobject"()
addStudentIt(1, ID, Curriculum) and setInterfaceAgent()
addNew(ID, name, ...)
startLess()
sendMessage(ACLMessage.AGREE)
sendMessage(ACLMessage.INFORM - content: OK)
requestResponse(requisita que o token seja enviado a RdP de 2o nível)
addNew(ID, name, ...)
startLess()
requestResponse(algum token requisitou uma resposta da interface externa)
<<create>> e configura a Mensagem para envio ao Agente Externo requisitando serviço de interface()
addStudent(2, ID, Problem)
Envia uma mensagem REQUEST para criar um novo token na RPO()
Envia mensagem AGREE()
Envia mensagem INFORM()

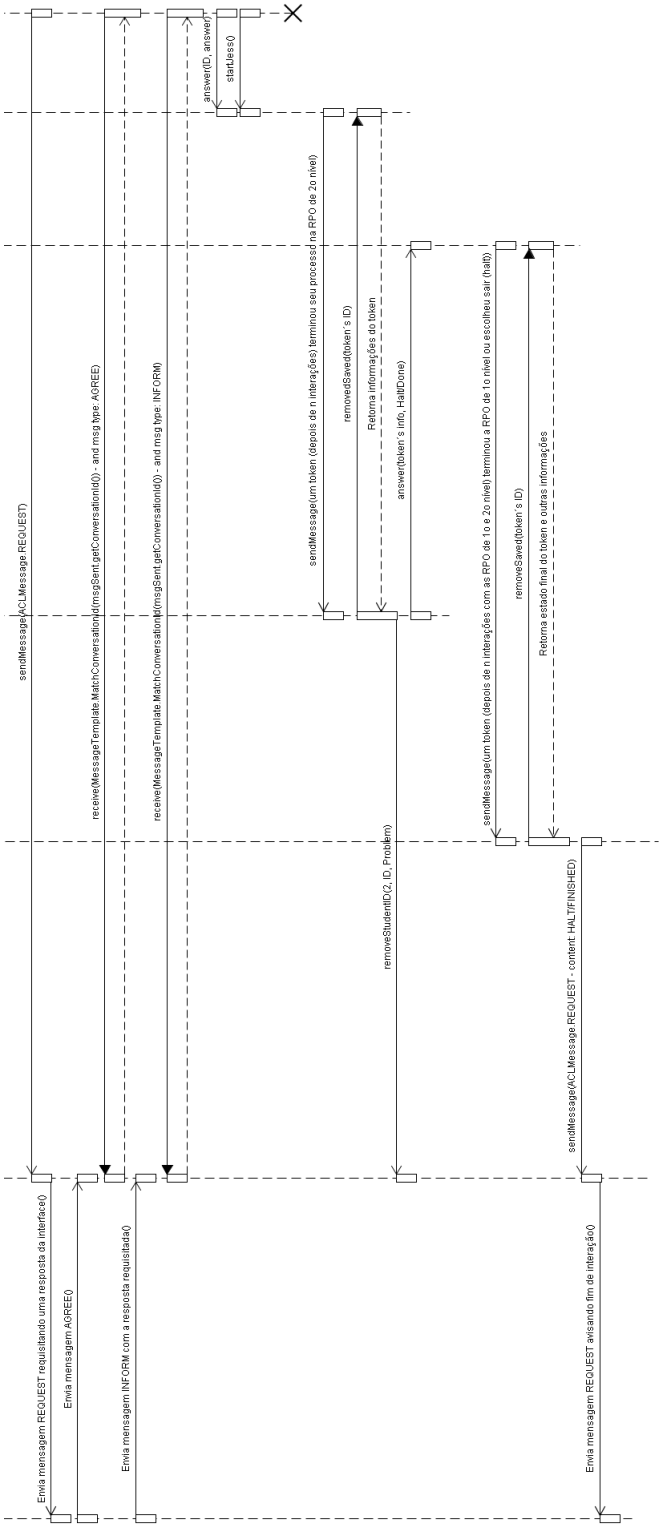
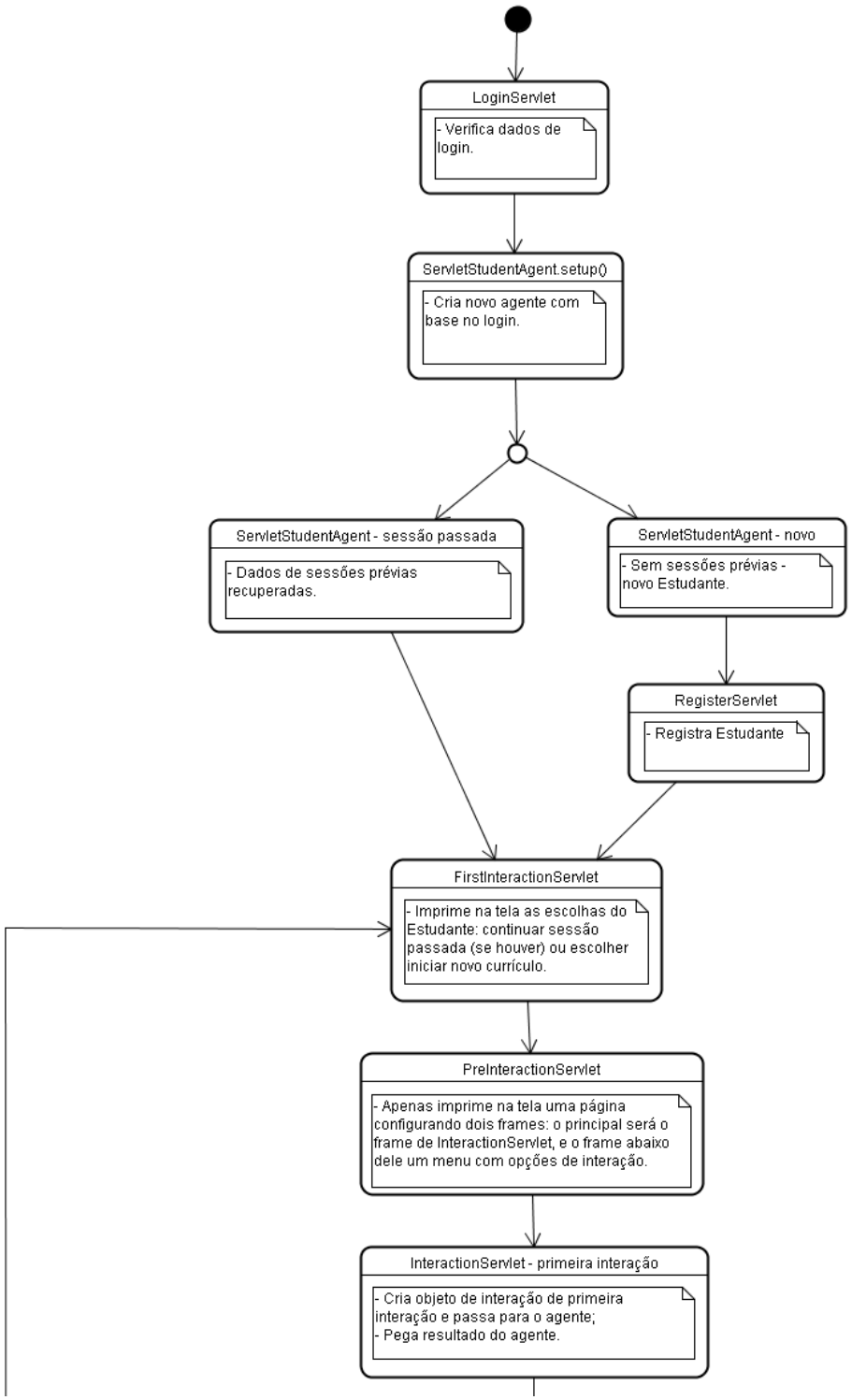



Figura D.2: Diagrama de Seqüência do agente StudentPetriNetsAgent


LoginServlet
- Verifica dados de login.
ServletStudentAgent.setup()
- Cria novo agente com base no login.
ServletStudentAgent - sessão passada
- Dados de sessões prévias recuperadas.
ServletStudentAgent - novo
- Sem sessões prévias - novo Estudante.
RegisterServlet
- Registra Estudante
FirstInteractionServlet
- Imprime na tela as escolhas do Estudante: continuar sessão passada (se houver) ou escolher iniciar novo currículo.
PreInteractionServlet
- Apenas imprime na tela uma página configurando dois frames: o principal será o frame de InteractionServlet, e o frame abaixo dele um menu com opções de interação.
InteractionServlet - primeira interação
- Cria objeto de interação de primeira interação e passa para o agente;
- Pega resultado do agente.

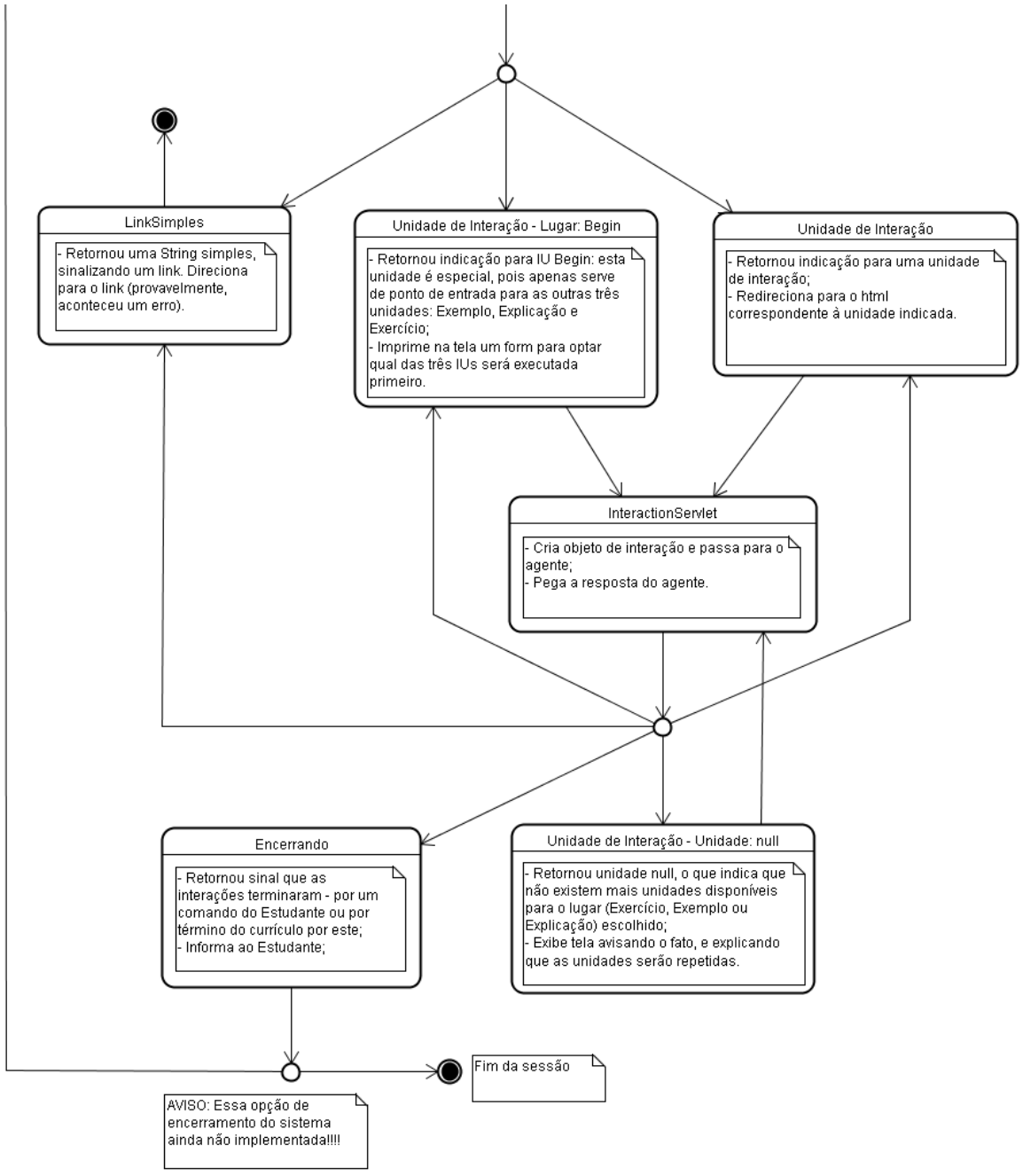



Figura D.3: Diagrama de Estados do agente StudentPetriNetsAgent

# Apêndice E

# Implementação da interface Web de Usuário do sistema MathTutor: Agente em JADE e Servlets

Um dos módulos do sistema MathTutor é a interface do estudante (ver figura 4.1). Neste trabalho esta interface foi desenvolvida para ser acessada via Web, como já citado em 5.4. Neste apêndice são apresentados detalhadamente a implementação dos dois elementos que compõem o controle da interface: o agente de interface e os Servlets.

## E.1 Agente JADE

Além dos agentes tutores da SATA (ver figura 4.1), outra classe de agente necessária no sistema multiagente são os agentes de interface. Como a própria designação sugere, eles fazem a ligação entre a SATA e o usuário estudante.

No sistema desenvolvido, a interface com o estudante é feita via tecnologias Web. Para prover também o dinamismo necessário à aplicação, o servidor Tomcat foi escolhido, bem como a tecnologia Servlets.

O agente de interface foi desenvolvido na classe ServletStudentAgent, no pacote `br.ufsc.MathTutor.ServletAgent`.

Entre os atributos desta classe, temos um objeto da classe `br.ufsc.MathTutor.JessComm.OpnStudent` que serve como modelo de dados do estudante:

```
private OpnStudent student;
```

Além disso, há também a referência para outro agente, que é o agente que contém as RPO do modelo pedagógico:

```
private AID opnAgent = null;
```

E também uma lista de currículos disponíveis no sistema:

```
private String curriculum[];
```

Reparando nas definições das estruturas acima, observa-se que elas não são inicializadas. Isso porque sua inicialização exige conhecimento sobre o sistema multiagente (o identificador de um agente que contém RPOs, e uma lista de currículos disponíveis). Como essas informações não são embutidas *a priori* nos agentes, elas precisam ser buscadas no sistema.

Para isso, usa-se o agente `DF` (Directory Facilitator), um agente padrão na arquitetura JADE, e que funciona como um serviço de "páginas amarelas", ou seja, um centro onde os agentes registram serviços por eles oferecidos, e/ou procuram por agentes que ofereçam determinados serviços.

No sistema desenvolvido, os agentes que contém as RPOs com o modelo pedagógico se registram no `DF` oferecendo um serviço do tipo "`PetriNets`" e com o nome do serviço sendo o nome do currículo. Assim, ao iniciar um agente `ServletStudentAgent`, este deve realizar uma busca no `DF` a fim de preencher a lista de currículos disponíveis. Este procedimento é implementado no *Behaviour* `GetInfoBehaviour`, uma *inner class* de `ServletStudentAgent`:

```
private class GetInfoBehaviour extends TickerBehaviour
```

O procedimento de busca no `DF` é simples, visto que o JADE provê métodos que facilitam a interação com este agente. Entretanto, como essa comunicação com o `DF` não está livre de falhas, o *Behaviour* implementado executa de tempos em tempos até conseguir uma resposta válida (a razão do *Behaviour* implementado ser subclasse de `TickerBehaviour`), até que ele obtenha o resultado esperado.

Além deste, há outro Behaviour implementado como *inner class* de `ServletStudentAgent`:

```
private class Servlet2AgentHandlerBehaviour extends CyclicBehaviour
```

Este *Behaviour*, que é executado ciclicamente, trata das requisições vindas dos Servlets. Isso é necessário por causa do modelo de trabalho do JADE.

O agente reside dentro do espaço de trabalho do Tomcat, ou seja, dentro da mesma máquina virtual (JVM) do Tomcat. Do ponto de vista dos Servlets, o agente é apenas mais um objeto. Entretanto, não é possível simplesmente de dentro dos Servlets, chamar algum método da classe do agente, como outro objeto qualquer, pois ao criar o agente, a plataforma JADE apenas retorna um "*proxy*" para o agente. Isto porque pode haver interferência na dinâmica de execução dos agentes, pois isto pode

causar problemas de sincronização de dados(*deadlocks*), em conjunto com a execução do escalonador interno que o mesmo possui para executar os seus `Behaviour`s.

Uma vez que o paradigma dos agentes em JADE é receber/enviar mensagens assincronamente, chamadas de métodos diretamente da classe do agente estariam fora desse escopo. Mesmo assim, é possível que classes Java não-agentes interajam com agentes. Para isso, os agentes JADE possuem uma opção de receber objetos Java como se estes fossem mensagens, através de uma fila específica. Esta opção não é padrão, e deve ser explicitamente feita pelo agente. No `ServletStudentAgent`, ela é feita dentro do método `setup()`, criando a fila com capacidade de 5 elementos:

```
this.setEnabledO2ACommunication(true, 5);
```

Para retirar um objeto desta fila, o método usado é:

```
Object obj = getO2AObject();
```

Entretanto, como geralmente necessita-se enviar dados diferenciados ao agente, foi definida uma classe que encapsula todos os dados necessários na comunicação Servlet/agente. Esta classe é `br.ufsc.MathTutor.ServletAgent.StudentInteraction`.

A classe `StudentInteraction` possui um campo que indica que tipo de interação o Servlet está iniciando com o agente:

```
private int type;
```

Por enquanto, as interações podem ser de quatro tipos: *login* do estudante, registro de dados do estudante, escolha de currículo do estudante, e interação padrão do estudante com o sistema executando as unidades de interação de cada problema do currículo:

```
public static final int LOGIN = 1;
public static final int REGISTER = 2;
public static final int FIRSTCHOICE = 3;
public static final int INTERACTION = 4;
```

Além destes, a classe `StudentInteraction` possui mais quatro propriedades abaixo listadas:

```
private HttpServletRequest theRequest = null;
private HttpServletResponse theResponse = null;
private boolean theResponseChangeFlag = false;
private Object data;
```

sendo que as duas primeiras, são objetos Java usados para manipular requisições e respostas do protocolo HTTP, e são fortemente ligadas aos Servlets. Entretanto, o agente tem acesso a estes objetos, podendo retirar dados diretamente da requisição HTTP, bem como manipular a resposta da requisição. Todavia, foi escolhido um modelo de trabalho no qual o agente não manipula diretamente a resposta ao cliente, mas passa dados ao Servlet (pelo propriedade data, de StudentInteraction), para que este devolva a resposta ao cliente. Este esquema, é mostrado na figura E.1, de maneira simplificada.

Os dados que são enviados ao Servlet pelo agente podem ser basicamente de dois tipos: uma String simples, contendo um endereço para o qual o usuário deve ser redirecionado; ou um vetor de Strings, contendo dados que serão usados pelo Servlet para montar uma página HTML para o usuário.

Como já dito acima, é o *Behaviour* `Servlet2AgentHandlerBehaviour` que lida com as interações entre os Servlets e o agente. A seguir, examinamos mais detalhadamente as ações deste behaviour.

Nos casos em que a interação com o Servlet é de login ou de registro (tipo de interação `LOGIN` ou `REGISTER`), basta que o *Behaviour* respectivamente valide o login ou registre o estudante no sistema, e retorne um endereço para o qual o usuário será redirecionado.

No caso em que a interação é do tipo `FIRSTCHOICE`, o *Behaviour* pode ter duas opções de execução: caso o estudante ainda não tenha decidido se deseja continuar a sessão passada ou se não foi escolhido currículo ainda, o *Behaviour* envia um vetor de Strings para o Servlet, cada String sendo uma opção de currículo; caso o estudante já tenha escolhido em qual currículo ele irá prosseguir, deve-se iniciar a interação entre este agente e o agente que contém o modelo pedagógico implementado numa RPO.

Neste segundo caso, o *Behaviour* obtém o endereço do agente com o currículo escolhido, e inicia outro *Behaviour*, que trata desta interação inter-agentes. Este *Behaviour* é o `PetriNetInitiator`, e é detalhado mais a frente.

O último caso que `Servlet2AgentHandlerBehaviour` trata, é o da interação tipo `INTERACTION`, que é uma interação do Estudante já executando uma Unidade de Interação de um problema dentro de um currículo. Neste caso, o *Behaviour* `Servlet2AgentHandlerBehaviour` apenas repassa para outro *Behaviour*, denominado `PetriNetResponder`, os dados da interação com o Servlet.

`PetriNetInitiator` e `PetriNetResponder` são os dois Behaviours que tratam da comunicação do agente `ServletStudentAgent` com o agente das RPOs `StudentPetriNetsAgent` (ver figura 5.6, repetida abaixo, para melhor visualização). Para a comunicação, utiliza-se o protocolo FIPA-Request.

`PetriNetInitiator` inicia a comunicação com o agente `StudentPetriNetsAgent`, solicitando a este, a inclusão na RPO de uma ficha representando o presente Estudante. A resposta a esta solicitação é apenas uma resposta de confirmação, sem nenhum dado relevante.

`PetriNetResponder` é o *Behaviour* que lida com as requisições vindas do agente `StudentPetriNetsAgent`. Estas requisições trazem também informações acerca da ficha da RPO que representa o aluno, informações estas que o *Behaviour* usa para mostrar uma página para o Estudante.

`PetriNetResponder` tem quatro fases distintas de atuação:

1. Recebe mensagem do agente das RPOs;
2. Com as informações vindas deste agente, passa para o Servlet informações;
3. Espera resposta do usuário vinda através do Servlet;
4. Envia resposta do usuário recebida como resposta para o agente das RPOs.

Para uma visão geral do agente, com todos os seus *Behaviours*, apresenta-se a figura E.3.

## E.2 Servlets

Os Servlets, apesar de cada um deles oferecer particularidades, têm estruturas e comportamentos similares, apresentados anteriormente na figura E.1. Abaixo, apresentamos uma descrição mais detalhada do código.

Inicialmente, procura-se pelo objeto de sessão (`HttpSession`), que representa a sessão do usuário no sistema. Se não houver sessão (o estudante não efetuou o *login* no sistema ou a sessão teve o *time-out alcançado*), o Servlet redireciona o usuário para uma tela de Sessão expirada:

```
HttpSession session = request.getSession(false);
if (session == null) {
   request.getRequestDispatcher("sessionExpired.htm").
                                forward(request, response);
...
```

O objeto `HttpSession` guarda uma referência para um `AgentController`, um *proxy* para o agente JADE. Essa referência é resgatada no Servlet, pois este precisa interagir com o agente:

```
AgentController agentProxy =
          (AgentController)session.getAttribute("agent");
```

Em seguida, o Servlet cria um objeto de interação com o agente, e envia para este:

```
interact = new StudentInteraction(request, response,
                         StudentInteraction.INTERACTION);
agentProxy.putO2AObject(interact, AgentController.ASYNC);
```

Feito isto, o Servlet deve esperar pela resposta do agente. Para isso, o objeto `interact`, da classe `StudentInteraction`, proporciona um método bloqueante, que só é liberado depois que o agente termina seu processamento e tem a resposta para o Servlet:

```
interact.waitChangedResponse();
```

A seguir, o Servlet obtém a resposta dada pelo agente, via objeto de interação:

```
Object ret = interact.getData();
```

Este processo apresentado é a base genérica para os Servlets. A partir daí, cada Servlet se diferencia pelo uso que faz dos dados advindos do agente jade. Em cada etapa mostrada, cada Servlet pode ser ligeiramente diferente, ou ainda haver outras instruções adicionadas. Esta metodologia foi adaptada de [72].

Até agora foi mostrado o esquema básico de funcionamento de todos os Servlets. Entretanto, ainda não foi mostrado a inicialização do agente, nem como é conseguido o objeto *proxy* (`AgentController`) armazenado na sessão. Os procedimentos para isso são realizados dentro do Servlet que serve como portão de entrada no sistema, o denominado `LoginServlet`.

`LoginServlet` possui como um de seus atributos, um *proxy* para um *Container* de agentes JADE (*Container*, no modelo JADE, é onde os agentes ?vivem?, e uma plataforma JADE seria o universo do agente, e esta pode conter um ou mais Containers, distribuídos ou não):

```
private AgentContainer agentContainer;
```

Este *Container* é inicializado logo na inicialização do Servlet (método `init`):

```
agentContainer = Util.getLocalAgentContainer();
```

`Util` é uma classe que como o nome sugere, contém métodos não pertencentes especificamente a nenhuma outra classe desenvolvida, mas que são essenciais. Dentro de `getLocalAgentContainer()`, primeiro adquire-se uma instância (*singleton*) do objeto que representa a execução do ambiente JADE:

```
jade.core.Runtime jadeRuntime = jade.core.Runtime.instance();
```

Em seguida, cria-se um objeto `Profile`. Este objeto contém todas as opções de inicialização de um ambiente JADE (as mesmas que podem ser passadas por linha de comando, caso o JADE seja iniciado pela mesma), e podem ser mudadas pelo programador. Como as opções padrão são suficientes para o sistema atual, basta que se construa o objeto usando o construtor com parâmetro false, que indica que o *Container* de agentes não será único (ou seja, este deverá se conectar ao *Container* principal):

```
jade.core.Profile jadeProfile = new jade.core.ProfileImpl(false);
```

A seguir, basta criar o *Container* de agentes propriamente dito:

```
agentContainer = jadeRuntime.createAgentContainer(jadeProfile);
```

Iniciado o `LoginServlet`, tem-se também iniciado um *Container* JADE para os agentes. O próximo passo é iniciar os agentes neste *Container*. A criação de agentes é realizada depois do usuário efetuar o *login* no sistema.

Mesmo na inicialização do agente, necessita-se de um objeto de interação (`StudentInteraction`), pois é através deste que é sincronizado o agente e o Servlet:

```
StudentInteraction firstInter =
            new StudentInteraction(null, null, 0);
```

Como esse objeto de interação tem o objetivo puramente de sincronização, nenhum dado relevante é enviado através dele. A seguir, é usado o método `createNewAgent` da classe `AgentController`:

```
AgentController agentProxy =
      agentContainer.createNewAgent(login+"Agent",
      "br.ufsc.MathTutor.ServletAgent.ServletStudentAgent",
      new Object[] {login, firstInter});
```

O primeiro parâmetro passado para o método `createNewAgent` é uma `String` que representa o nome que o agente deverá ter no ambiente. No sistema desenvolvido, esse nome é composto pelo *login* do usuário mais a palavra ?Agent?. O segundo parâmetro é uma `String` contendo a classe Java do agente. O terceiro parâmetro é um vetor de `Object`s, contendo parâmetros para inicialização do agente. No sistema desenvolvido, são enviados uma `String` com o login do usuário, e o objeto de interação previamente inicializado.

Uma vez criado, o agente não começa a sua execução instantaneamente. Na verdade, nem mesmo o seu método `setup()` é chamado. É preciso explicitamente iniciar o agente, usando o método `start()` de `AgentController`. A seguir, o Servlet espera pela inicialização apropriada do agente (sincronização), usando o objeto de interação passado como parâmetro:

```
agentProxy.start();
firstInter.waitChangedResponse();
```

Tendo o agente sido inicializado sem problemas, o seu *proxy* é colocado no objeto de sessão:

```
session.setAttribute("agent", agentProxy);
```

O restante do procedimento de `LoginServlet` é parecido com o modelo padrão de ações dos Servlets, descritos inicialmente. Para visualizar o fluxo do usuário pelos Servlets, pode-se observar a figura E.4.

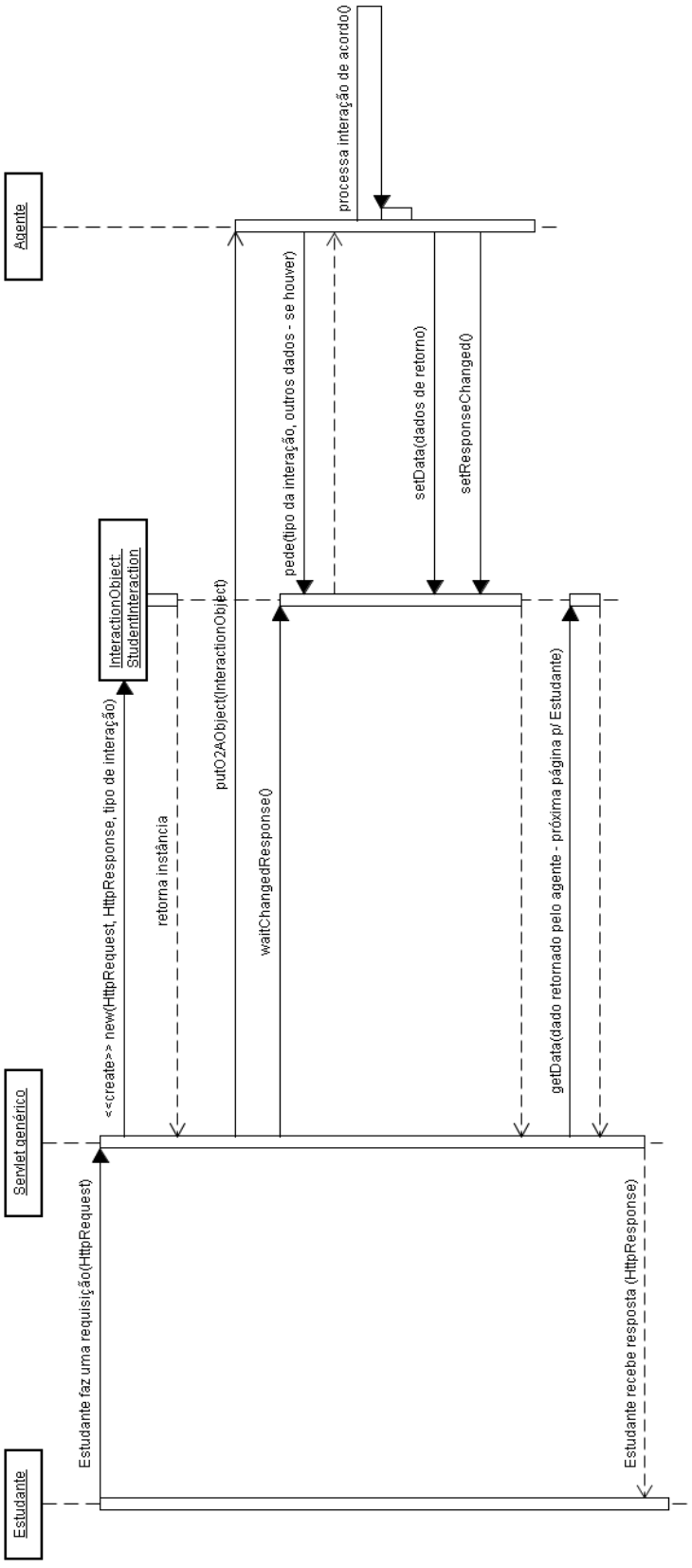



Figura E.1: Diagrama de Seqüência das interações Servlets/Agente

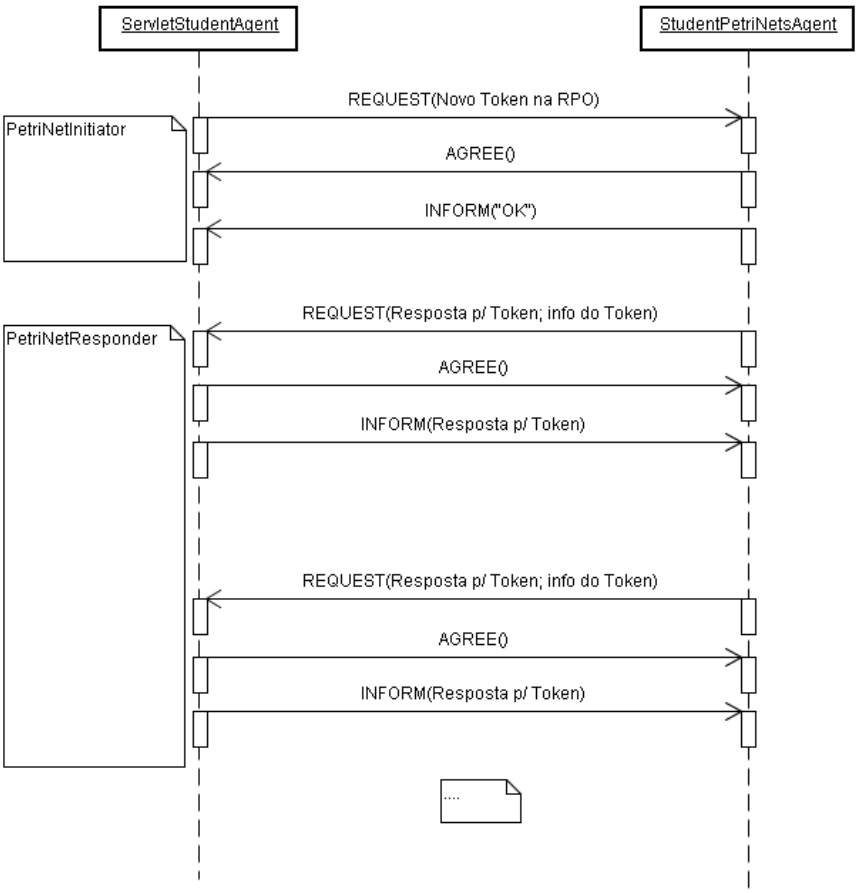
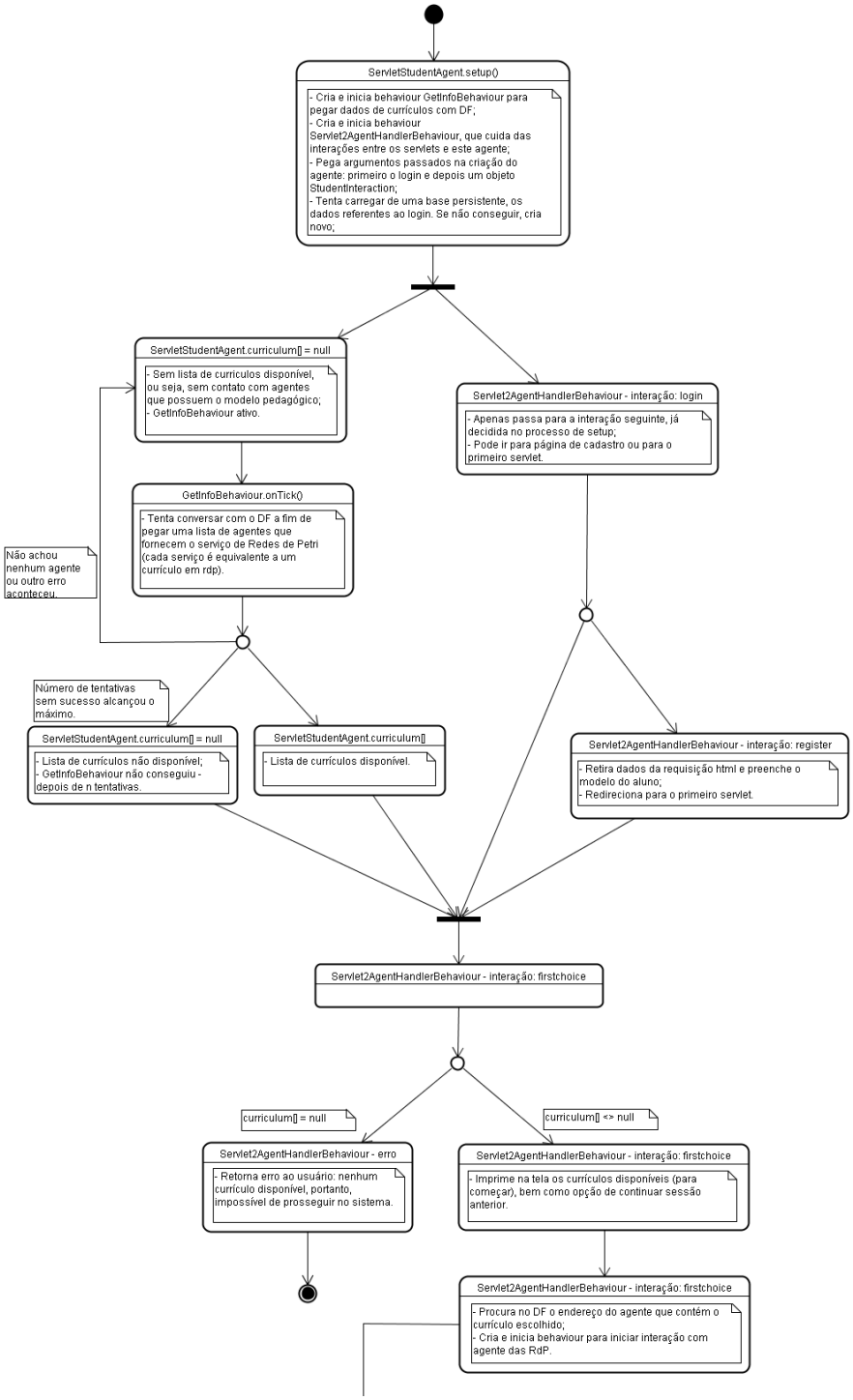



Figura E.2: Comunicação entre Agentes


ServletStudentAgent.setup()
- Cria e inicia behaviour GetInfoBehaviour para pegar dados de currículos com DF;
- Cria e inicia behaviour Servlet2AgentHandlerBehaviour, que cuida das interações entre os servlets e este agente;
- Pega argumentos passados na criação do agente: primeiro o login e depois um objeto StudentInteraction;
- Tenta carregar de uma base persistente, os dados referentes ao login. Se não conseguir, cria novo;
ServletStudentAgent.curriculum[] = null
- Sem lista de currículos disponível, ou seja, sem contato com agentes que possuem o modelo pedagógico;
- GetInfoBehaviour ativo.
Servlet2AgentHandlerBehaviour - interação: login
- Apenas passa para a interação seguinte, já decidida no processo de setup;
- Pode ir para página de cadastro ou para o primeiro servlet.
GetInfoBehaviour.onTick()
- Tenta conversar com o DF a fim de pegar uma lista de agentes que fornecem o serviço de Redes de Petri (cada serviço é equivalente a um currículo em rdp).
Não achou nenhum agente ou outro erro aconteceu.
Número de tentativas sem sucesso alcançou o máximo.
ServletStudentAgent.curriculum[] = null
- Lista de currículos não disponível;
- GetInfoBehaviour não conseguiu - depois de n tentativas.
ServletStudentAgent.curriculum[]
- Lista de currículos disponível.
Servlet2AgentHandlerBehaviour - interação: register
- Retira dados da requisição html e preenche o modelo do aluno;
- Redireciona para o primeiro servlet.
Servlet2AgentHandlerBehaviour - interação: firstchoice
curriculum[] = null
curriculum[] <> null
Servlet2AgentHandlerBehaviour - erro
- Retorna erro ao usuário: nenhum currículo disponível, portanto, impossível de prosseguir no sistema.
Servlet2AgentHandlerBehaviour - interação: firstchoice
- Imprime na tela os currículos disponíveis (para começar), bem como opção de continuar sessão anterior.
Servlet2AgentHandlerBehaviour - interação: firstchoice
- Procura no DF o endereço do agente que contém o currículo escolhido;
- Cria e inicia behaviour para iniciar interação com agente das RdP.

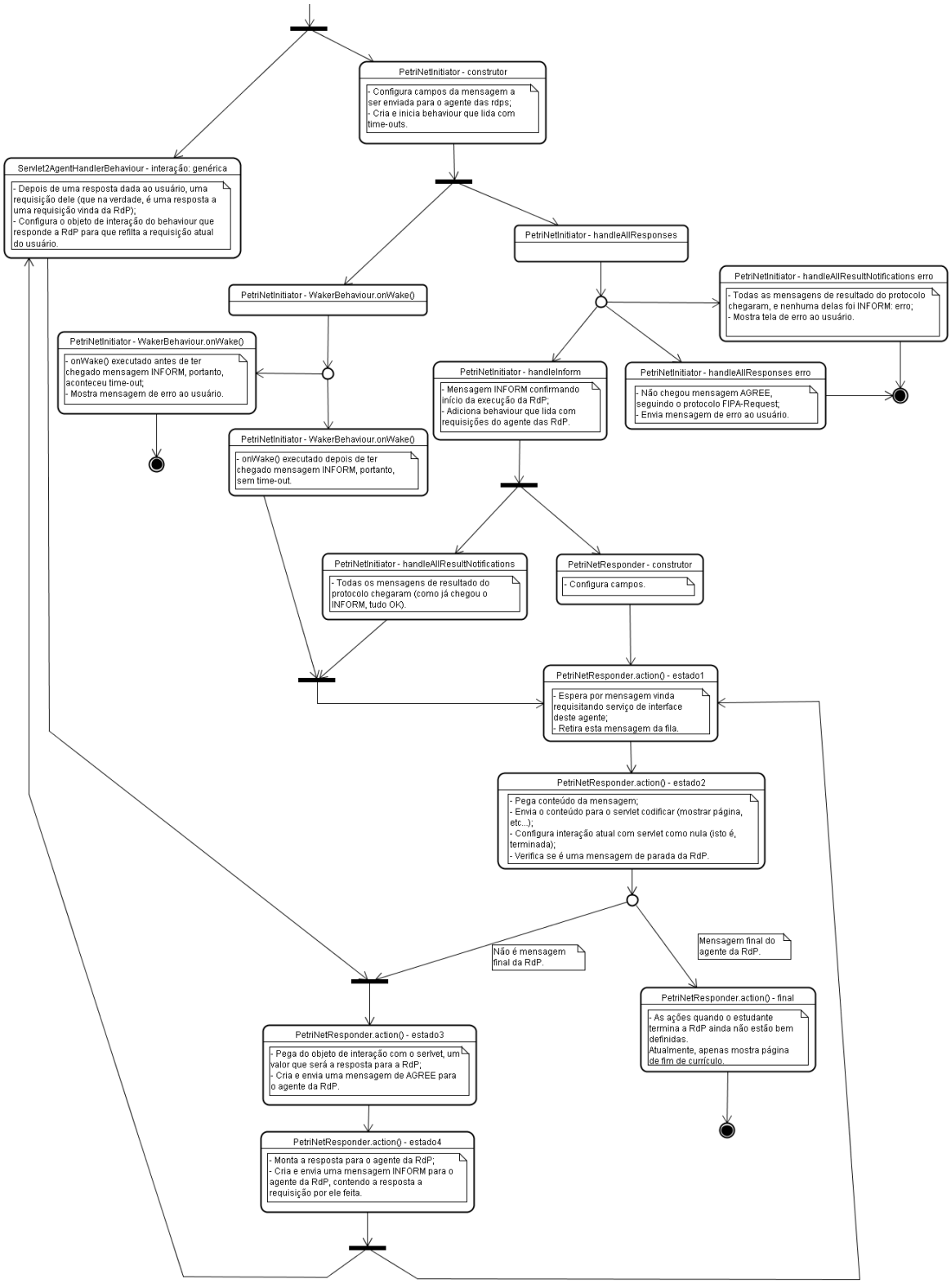
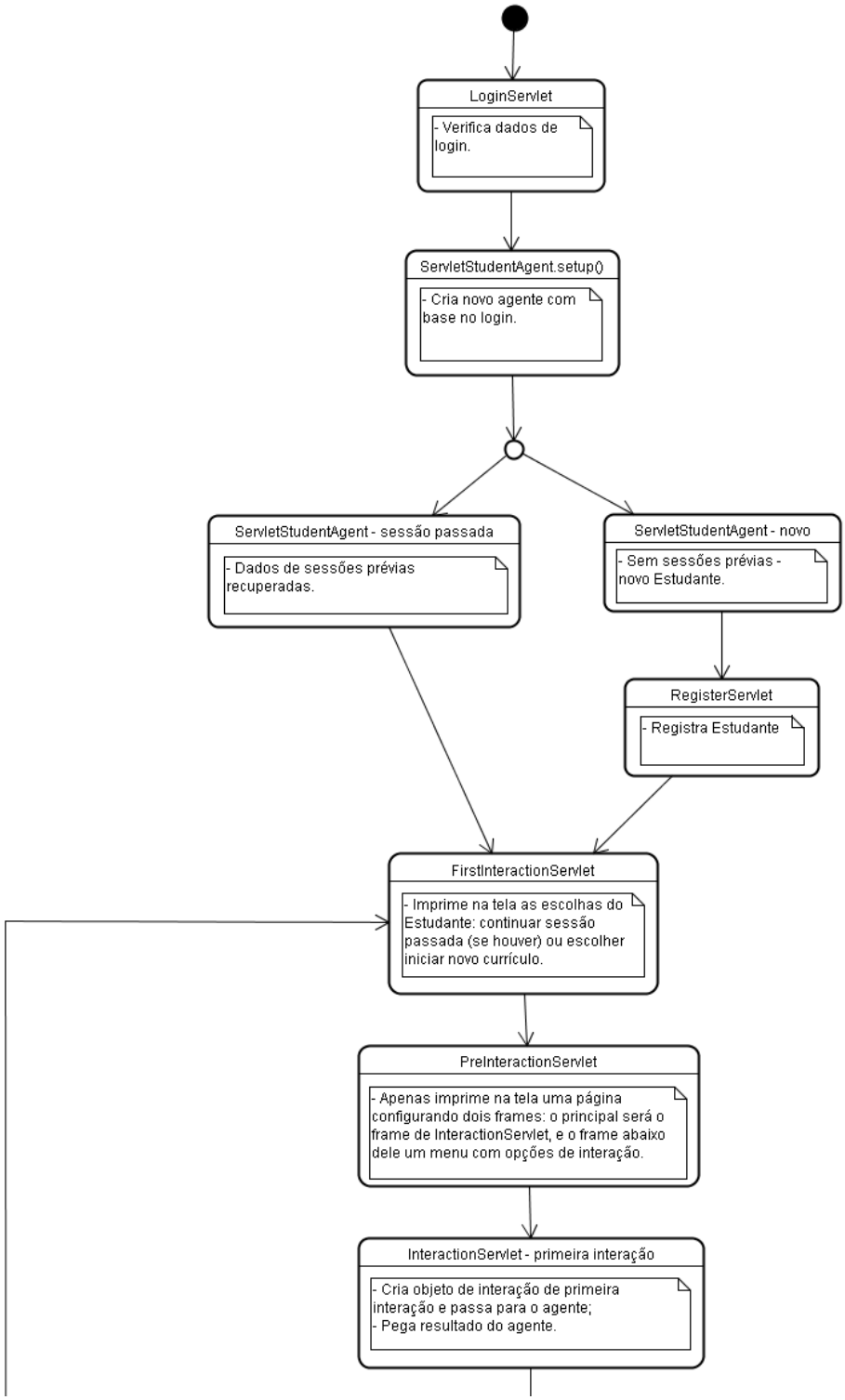



Figura E.3: Diagrama de Estados do pacote ServletAgent


LoginServlet
- Verifica dados de login.
ServletStudentAgent.setup()
- Cria novo agente com base no login.
ServletStudentAgent - sessão passada
- Dados de sessões prévias recuperadas.
ServletStudentAgent - novo
- Sem sessões prévias - novo Estudante.
RegisterServlet
- Registra Estudante
FirstInteractionServlet
- Imprime na tela as escolhas do Estudante: continuar sessão passada (se houver) ou escolher iniciar novo currículo.
PreInteractionServlet
- Apenas imprime na tela uma página configurando dois frames: o principal será o frame de InteractionServlet, e o frame abaixo dele um menu com opções de interação.
InteractionServlet - primeira interação
- Cria objeto de interação de primeira interação e passa para o agente;
- Pega resultado do agente.

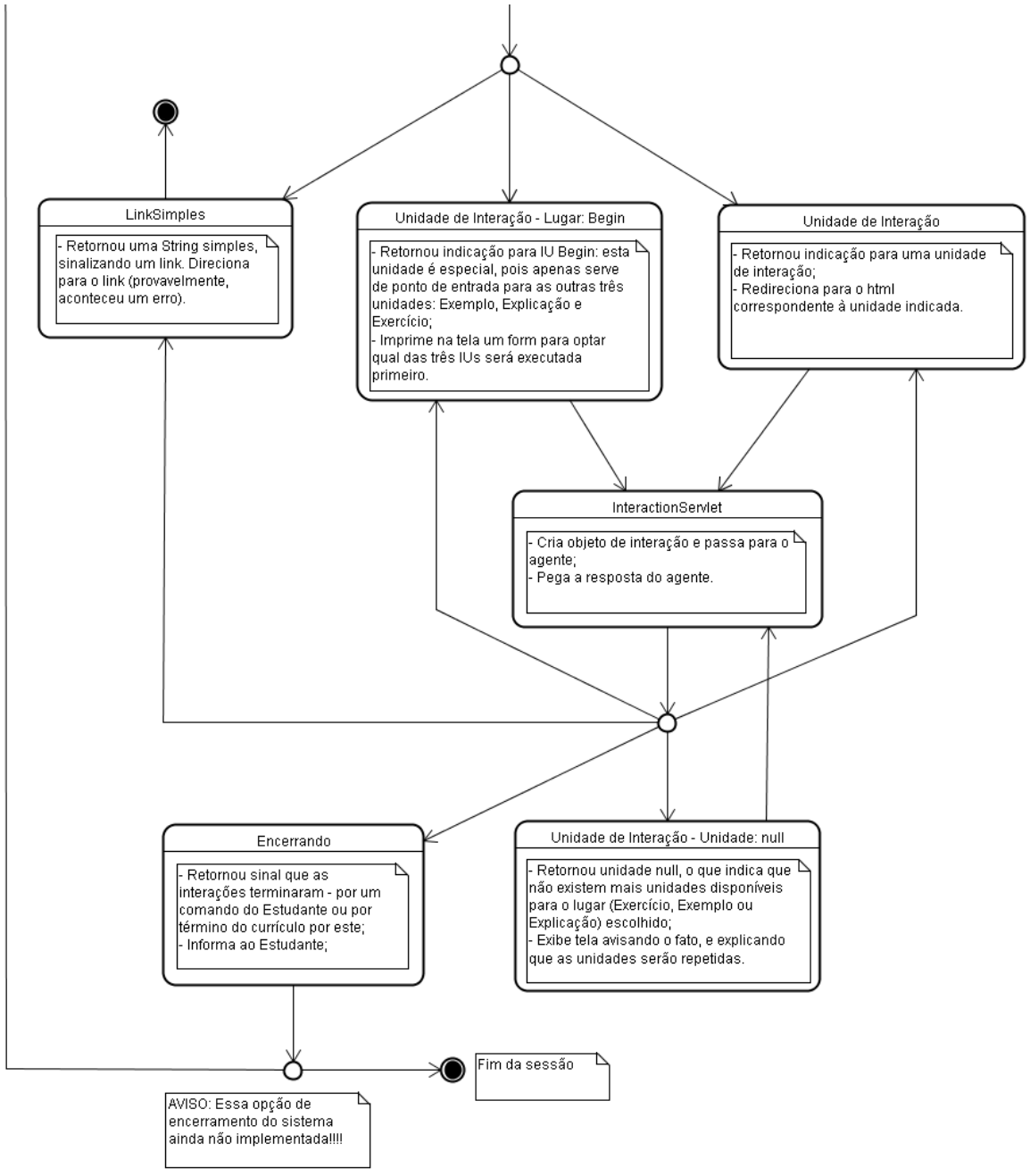



Figura E.4: Diagrama de Estados contendo o fluxo de execução dos Servlets

# Referências Bibliográficas


[1] Luciana Bolan Frigo. Mathtutor - um ambiente interativo multi-agente para o ensino de estrutura da informação. Master's thesis, Universidade Federal de Santa Catarina - Engenharia Elétrica - Florianópolis, SC, 2002.

[2] Adriana Postal. Navegação em um ambiente interativo multi-agentes para ensino. Master's thesis, Universidade Federal de Santa Catarina - Engenharia Elétrica - Florianópolis, SC, 2004.

[3] Evandro de Barros Costa. *Um Modelo de Ambiente Interativo de Aprendizagem Baseado numa Arquitetura Multi-Agentes*. PhD thesis, Universidade Federal da Paraíba - Departamento de Engenharia Elétrica - Campina Grande, PB, 1997.

[4] Stuart J. Russell and Peter Norvig. *AI: A Modern Approach*. Prentice Hall, 1995.

[5] Ricardo Rabelo. Notas de aula do prof. ricardo rabelo para disciplina inteligência artificial aplicada à controle e automação. http://www.das.ufsc.br/ Acessado em 02/2006.

[6] William Bolzan and Lúcia Maria Martins Giraffa. Estudo comparativo sobre sistemas tutores inteligentes multiagentes web. Technical report, Faculdade de Informática - PUCRS - Brazil, 2002.

[7] Randall Davis, Howard E. Shrobe, and Peter Szolovits. What is a knowledge representation. *AI Magazine*, 14(1):17–33, 1993.

[8] Guilherme Bittencourt. *Inteligência Artificial: Ferramentas e Teorias*. Editora da UFSC, 1998.

[9] Volney Gadelha Lustosa. O estado da arte em inteligência artificial. *Colabo@ - Revista Digital da CVA-Ricesu*, 2, número 8, 2004. Acessado em 02/2006.

[10] Shu-Hsien Liao. Expert system methodologies and applications-a decade review from 1995 to 2004. *Expert Systems with Applications*, 28, Issue 1:93–103, 2005.

[11] Jukka K. Nurminen, Olli Karonen, and Kimmo Hätönen. What makes expert systems survive over 10 years - empirical evaluation of several engineering applications. *Expert Systems with Applications*, 24, Issue 2:199–211, 2003.

[12] Edward H. Shortliffe. *Computer-Based Medical Consultations: MYCIN*. American Elsevier, New York, 1976.

[13] Lisp programming language - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Lisp_programming_language. Acessado em 02/2006.

[14] Mycin - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Mycin. Acessado em 02/2006.

[15] Thomas R. Gruber. Towards principles for the design of ontologies used for knowledge sharing. In N. Guarino and R. Poli, editors, *Formal Ontology in Conceptual Analysis and Knowledge Representation*, Deventer, The Netherlands, 1993. Kluwer Academic Publishers.

[16] Ontologia - wikipedia, the free encyclopedia. http://pt.wikipedia.org/wiki/Ontologia. Acessado em 02/2006.

[17] Ismênia de Oliveira, Rafael Robinson, and Rosario Girardi. Uma ontologia para a especificação de sistemas de padrões. In Ed. Rossana M. C. Andrade et al, editor, *Anais da Quarta Conferência Latino-Americana em Linguagens de Padrões para Programação - SugarLoaf-PLOP2004*, pages 281–292, Fortaleza - CE, Brasil, 2004.

[18] Asunción Gómes Pérez. Ontoweb deliverable 1.3 - a survey on ontology tools. http://ontoweb.org/About/Deliverables/D13_v1-0.zip, 2002. Acessado em 02/2006.

[19] Tutor - wikipedia, the free encyclopedia. http://en.wikipedia.org/wiki/Tutor. Acessado em 02/2006.

[20] Sidney pressey. http://www.ittheory.com/pressey.htm. Acessado em 02/2006.

[21] John Self. Theoretical foundations for intelligent tutoring systems, 1990.

[22] John A. Self. The defining characteristics of intelligent tutoring systems research: Itss care, precisely. *International Journal of Artificial Intelligence in Education*, 10:350–364, 1999.

[23] Marta Costa Rosatelli. Novas tendências da pesquisa em inteligência artificial na educação. In R. C. Nunes, editor, *VIII Escola de Informática da SBC Sul*, pages 179–210. Editora da UFRGS, 2000.

[24] Eliane Pozzebon and Jorge Muniz Barreto. Inteligência artificial no ensino com tutores inteligentes. *Revista de divulgação científica e cultural*, v. 5, número 1 e 2:141–162, Dezembro 2002.

[25] Eliane Pozzebon, Guilherme Bittencourt, and Janette Cardoso. Uma arquitetura multiagente para suporte ao aprendizado em grupo em sistemas tutores inteligentes. In *Proceedings of XVI Simpósio Brasileiro de Informática na Educação - SBIE 2005*, 2005.

[26] Yujian Zhou and Martha W. Evens. A practical student model in an intelligent tutoring system. In *ICTAI*, pages 13–18, 1999.

[27] Yujian Zhou. *Building a New Student Model to Support Adaptive Tutoring in a Natural Language Dialogue System*. PhD thesis, Illinois Institute of Technology, 2000.

[28] Martha W. Evens, Stefan Brandle, Ru-Charn Chang, Reva Freedman, Michael Glass, Yoon Hee Lee, Leem Seop Shim, Chong Woo Woo, Yuemei Zhang, Yujian Zhou, Joel A. Michael, and Allen A. Rovick. Circsim-tutor: An intelligent tutoring system using natural language dialogue. In *Twelfth Midwest AI and Cognitive Science Conference, MAICS 2001, Oxford, OH*, pages 16–23., 2001.

[29] Breno F. T. Azevedo. Mutantis - um sistema tutor inteligente multi-agente para o ensino - aprendizagem de conceitos. In *X Simpósio Brasileiro de Informática na Educação*, 1999.

[30] Mihal Badjonski, Mirjana Ivanovic, and Zoran Budimac. Intelligent tutoring system as multiagent system. In *Proceedings of ICPIS '97 (IEEE International Conf. on Intelligent Processing Systems)*, pages 871–875, Beijing, China, 1997.

[31] Maria Virvou and George Katsionis. Web services for an intelligent tutoring system that operates as a virtual reality game. In *Proceedings of the IEEE International Conference on Systems, Man and Cybernetics (SMC 2003)*, Washington D.C, 2003.

[32] B. Roche, J. Kilbride, and E. Mangina. Analysis and design of an agent-based intelligent tutoring system for e-learning within irish universities. In *Proceedings of International Conference on Information and Communication Technologies: From Theory to Applications*, Abril 2004.

[33] D. Gürer. The use of distributed agents in intelligent tutoring. In *Proceedings of Workshop On Pedagogical Agents, ITS'98 (San Antonio)*, pages 20–25, 1998.

[34] Tom Murray. Authoring intelligent tutoring systems: An analysis of the state of the art. In *International Journal of Artificial Intelligence in Education*, volume 10, pages 98–129, 1999.

[35] Janete Cardoso and Robert Valette. *Redes de Petri*. Série Didática. Editora da UFSC - Florianópolis, SC, 1 edition, 1997.

[36] Charles A. Lakos. From coloured petri nets to object petri nets. In *Proceedings of the Application and Theory of Petri Nets 1995*, volume 935, pages 278–297, Berlin, Germany, 1995. Springer-Verlag.

[37] Charles A. Lakos. The object orientation of object petri nets. In *Proceedings of the first international workshop on Object-Oriented Programming and Models of Concurrency - 16th International Conference on Application and Theory of Petri Nets*, pages 1–14, 1995.

[38] James Gosling and Henry McGilton. The java language environment white paper. http://java.sun.com/docs/white/langenv/, 1996.

[39] Java technology. http://java.sun.com.

[40] java.net - the source for java technology collaboration. www.java.net. Acessado em 02/2006.

[41] Sourceforge.net. http://sourceforge.net/. Acessado em 02/2006.

[42] Javacc home. https://javacc.dev.java.net/. Acessado em 02/2006.

[43] Backus-naur form. http://en.wikipedia.org/wiki/Backus-Naur_form. Acessado em 02/2006.

[44] Jim Hugunin. Python and java - the best of both worlds. In *Proceedings of the 6th International Python Conference*, 1997.

[45] Viswanathan Kodaganallur. Incorporating language processing into java applications: a javacc tutorial. *IEEE Software*, 21(4):70–77, 2004.

[46] Jade - java agent development framework. http://jade.tilab.com/.

[47] Fabio Bellifemine, Giovanni Caire, A. Poggi, and Giovanni Rimassa. Jade a white paper. http://jade.tilab.com/papers/2003/WhitePaperJADEEXP.pdf. Acessado em 02/2006.

[48] Fipa - foundation for intelligent physical agents. http://www.fipa.org/.

[49] GNU. Gnu lesser general public license. http://www.gnu.org/copyleft/lesser.html. Acessado em 02/2006.

[50] Krzysztof Chmiel, Maciej Gawinecki, Pawel Kaczmarek, Michal Szymczak, and Marcin Paprzycki. Efficiency of jade agent platform, 2005.

[51] Fabio Bellifemine, Giovanni Caire, Tiziana Trucco, and Giovanni Rimassa. Jade programmer's guide. http://jade.tilab.com/doc/programmersguide.pdf, 2005. Acessado em 02/2006.

[52] Giovanni Caire. Jade tutorial - jade programming for beginners. http://jade.tilab.com/doc/JADEProgramming-Tutorial-for-beginners.pdf, 2003. Acessado em 02/2006.

[53] Fabio Bellifemine, Giovanni Caire, Tiziana Trucco, Giovanni Rimassa, and Roland Mungenast. Jade administrator's guide. http://jade.tilab.com/doc/administratorsguide.pdf, 2005. Acessado em 02/2006.

[54] Jess, the rule engine for the java platform. http://herzberg.ca.sandia.gov/jess/.

[55] Clips: A tool for building expert systems. http://www.ghg.net/clips/CLIPS.html.

[56] Charles L. Forgy. Rete: A fast algorithm for the many pattern/ many object pattern match problem. *rtificial Intelligence*, 19:17–37, 1982.

[57] The protégé ontology editor and knowledge acquisition system. http://protege.stanford.edu/. Acessado em 02/2006.

[58] Holger Knublauch. An ai tool for the real world - knowledge modeling with protégé. http://www.javaworld.com/javaworld/jw-06-2003/jw-0620-protege.html, Junho 2003. Acessado em 02/2006.

[59] ACM SIGCHI. Acm sigchi curricula for human-computer interaction. http://sigchi.org/cdg/index.html. Acessado em 02/2006.

[60] Brad A. Myers. Why are human-computer interfaces difficult to design and implement? Technical Report CMU-CS-93-183, School of Computer Science, Carnegie-Mellon University, Julho 1993.

[61] W3C. W3c html home page. http://www.w3.org/MarkUp/. Acessado em 02/2006.

[62] W3C. World wide web consortium. http://www.w3.org/. Acessado em 02/2006.

[63] W3C. Cgi: Common gateway interface. http://www.w3.org/CGI/. Acessado em 02/2006.

[64] Sun. Java servlet technology. http://java.sun.com/products/servlet/. Acessado em 02/2006.

[65] Apache Software Foundation. Apache tomcat. http://tomcat.apache.org/. Acessado em 02/2006.

[66] Marty Hall. *Core Servlets and JavaServer Pages*. Prentice Hall, 2000.

[67] Sun. Javaserver pages technology. http://java.sun.com/products/jsp/. Acessado em 02/2006.

[68] Projeto mathnet. http://mathnet.lcmi.ufsc.br. Acessado em 02/2006.

[69] Guilherme Bittencourt. Fundamentos da estrutura da informação. http://www.das.ufsc.br/ gb/fei. Acessado em 02/2006.

[70] Janétte Cardoso, Guilherme Bittencourt, Luciana Bolan Frigo, and Eliane Pozzebon. Petri nets for authoring mechanism. In EDUA Editora da Universidade Federal do Amazonas, editor, *Anais do XV SBIE - Simpósio Brasileiro de Informática na Educação*, volume 1, pages 378–387, Manaus-AM, 2004.

[71] Petri nets tools and softwares. http://www.informatik.uni-hamburg.de/TGI/PetriNets/tools/. Acessado em 02/2006.

[72] Fabien Gandon. Linking a servlet to a jade agent. http://jade.tilab.com/community-3rdpartysw.htm, 2003. Acessado em 02/2006.